Parallel K-clique Counting on GPUs

Mohammad Almasri
almasri3@illinois.edu
University of Illinois at Urbana-Champaign

Izzat El Hajj
izzat.elhajj@aub.edu.lb
American University of Beirut

Rakesh Nagi
nagi@illinois.edu
University of Illinois at Urbana-Champaign

Jinjun Xiong
jinjun@buffalo.edu
University at Buffalo

Wen-mei Hwu
w-hwu@illinois.edu
University of Illinois at Urbana-Champaign, NVIDIA

ABSTRACT

Counting k-cliques in a graph is an important problem in graph analysis with many applications such as community detection and graph partitioning. Counting k-cliques is typically done by traversing search trees starting at each vertex in the graph. Parallelizing k-clique counting has been well-studied on CPUs and many solutions exist. However, there are no performant solutions for k-clique counting on GPUs.

Parallelizing k-clique counting on GPUs comes with numerous challenges such as the need for extracting fine-grain multi-level parallelism, sensitivity to load imbalance, and constrained physical memory capacity. While there has been work on related problems such as finding maximal cliques and generalized sub-graph matching on GPUs, k-clique counting in particular has yet to be explored in depth. In this paper, we present the first parallel GPU solution specialized for the k-clique counting problem. Our solution supports both graph orientation and pivoting for eliminating redundant clique discovery. It incorporates both vertex-centric and edge-centric parallelization schemes for distributing work across thread blocks, and further partitions work within each thread block to extract fine-grain multi-level parallelism while tolerating load imbalance. It also includes optimizations such as binary encoding of induced sub-graphs and sub-warp partitioning to limit memory consumption and improve the utilization of execution resources.

Our evaluation shows that our best GPU implementation outperforms the best state-of-the-art parallel CPU implementation by a geometric mean of 12.39×, 6.21×, and 18.99× for $k = 4$, 7, and 10, respectively. We also perform a detailed evaluation of the trade-offs involved in the choice of parallelization scheme, and the incremental speedup of each optimization to provide an in-depth understanding of the optimization space. The insights from our optimization flow can be useful for optimizing other clique finding and graph mining solutions on GPUs. Our code will be open-sourced to enable further research on GPU parallelization of k-clique counting and other similar graph mining algorithms.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

GPU, graphs, k-clique counting, parallel search tree traversal

ACM Reference Format:
Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. Parallel K-clique Counting on GPUs. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3524059.3532382

1 INTRODUCTION

Dense sub-graph counting and listing is an important problem in graph mining [22, 37]. A k-clique (or a k-vertex clique) in a graph is a complete sub-graph with exactly k vertices and $k \times (k-1)$ edges, such that every vertex in the clique is connected to every other vertex. Counting k-cliques is a useful algorithmic component of solutions to many problems such as community detection [19, 27, 58, 72], graph partitioning and compression [23, 52, 53], learning network embedding [51, 71], and recommendation systems [45, 60].

A common approach to k-clique counting is to traverse a search tree for each vertex and find k-cliques that contain that vertex. This approach is commonly parallelized by processing different trees or subtrees in parallel. One fundamental optimization is to eliminate the search tree branches that discover the same clique redundantly. Two prominent approaches to this optimization are graph orientation [11, 13, 40, 56] and pivoting [33].

The graph orientation approach transforms the graph into a directed graph so that each k-clique, which is a symmetric structure, is only found from one of the vertices it contains. Common orientation criteria include vertex degree [11, 20, 56], graph coloring [40], and degeneracy based on k-core decomposition [13, 18, 56] or relaxations of k-core [56]. The pivoting approach [33] for k-clique counting is inspired by the Bron-Kerbosh maximal clique finding approach [18]. Rather than searching for all k-cliques, the pivoting approach finds the largest cliques, then calculates the number of k-cliques they contain. To the best of our knowledge, the state-of-the-art parallel implementations for the graph orientation and pivoting approaches to k-clique counting are ARB-COUNT [56] and Pivoter [33], respectively. Both of these parallel implementations are designed for CPUs.

The massively parallel hardware resources in modern GPUs offer promising opportunities for accelerating k-clique counting

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532382

for large graphs. However, successful parallelization of k-clique counting on GPUs must overcome the additional challenges arising from the differences in hardware characteristics between GPUs and CPUs. The first major challenge is that GPUs require more fine-grain parallelism to be extracted from the computation to utilize parallel hardware resources efficiently. These parallel hardware resources are organized into a multi-level hierarchy which further complicates the parallelization process. Moreover, the massively parallel nature of the hardware makes GPUs more sensitive to load imbalance. The second major challenge is that GPUs come with faster but smaller physical memories than CPUs. The limited GPU memory capacity can severely limit parallelism because there may not be sufficient memory for tracking the execution state of the large number of threads that traverse search trees and subtrees in parallel. While there has been work on solving related problems on GPUs, such as finding maximal cliques [30, 34, 39, 59, 67, 70] and generalized sub-graph matching [9, 28, 63], little attention has been given to k-clique counting in particular. To the best of our knowledge, there are no performant parallel solutions specialized for k-clique counting on GPUs.

In this paper, we propose a novel parallel GPU solution to the k-clique counting problem. Our proposed solution supports both graph orientation and pivoting for eliminating redundant clique discovery. It incorporates both vertex-centric and edge-centric parallelization schemes for distributing work across thread blocks, as well as different ways for partitioning work within each thread block to extract fine-grain multi-level parallelism while tolerating load imbalance. It uses binary encoding of induced sub-graphs to drastically reduce memory consumption while allowing for highly parallel list intersection operations. It also takes advantage of the new independent thread scheduling support in recent GPUs (Volta and beyond) to allow threads to collaborate at sub-warp granularity for more effective multi-level parallelization and better utilization of parallel execution resources. It further employs various other techniques to limit memory consumption. Finding the right combination of optimizations that work together effectively is a key contribution of our work, and we believe that the insights from our optimization flow can be useful for optimizing other clique finding and graph mining solutions on GPUs.

Our evaluation shows that our best GPU implementation significantly outperforms the best state-of-the-art parallel CPU implementation by a geometric mean of 12.39×, 6.21×, and 18.99× for $k = 4$, 7, and 10, respectively. Our parallel solution scales to graphs with billions of edges for arbitrary values of k. We perform a detailed evaluation of the trade-offs between the vertex-centric and the edge-centric parallelization schemes, particularly pertaining to their impact on load imbalance and their interaction with the two redundancy elimination approaches and different values of k. We also show that binary encoding improves performance by a 2.17× and 1.38× and that sub-warp partitioning improves performance by 1.98× and 1.73× for graph orientation and pivoting, respectively.

2 BACKGROUND

2.1 Clique Counting

A common approach to counting k-cliques in a graph is to traverse a search tree for each vertex in the graph to find k-cliques that contain that vertex. The search tree for each vertex (1-clique) branches out to the vertex's neighbors to find edges (2-cliques), then for each edge, branches out to the common neighbors of the edge's endpoints to find triangles (3-cliques), then for each triangle, branches out to the common neighbors of the vertices in the triangle to find 4-cliques, and so on. In general, for each $(k-1)$-clique, the tree branches out to the common neighbors of the $k-1$ vertices in the clique to find the k-cliques. This approach to k-clique counting is commonly parallelized by processing different trees or subtrees in parallel.

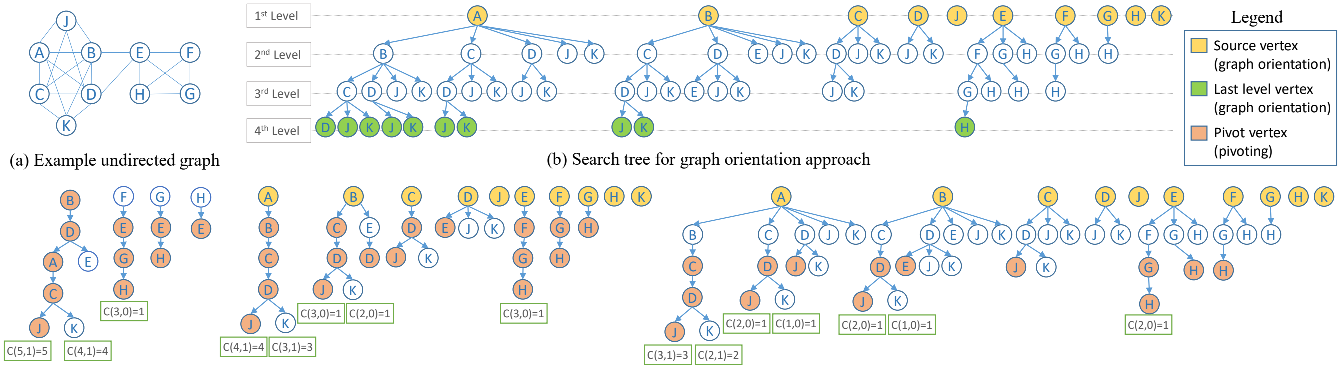
One key distinguishing feature among algorithms is how they avoid discovering the same clique redundantly from multiple root vertices. Avoiding redundant clique discovery results in a substantial reduction in the amount of work done, thereby improving performance. The two major approaches to avoiding redundant clique discovery are: (1) orienting the graph before traversal, and (2) pivoting. These two approaches are described in Sections 2.2 and 2.3, respectively.

2.2 Graph Orientation Approach

Graph orientation (or vertex ordering) is a preprocessing step that transforms the graph from an undirected graph to a directed one. Common orientation criteria include vertex degree [11, 20, 56], graph coloring [40], and degeneracy based on k-core decomposition [13, 18, 56] or relaxations of k-core [56]. Graph orientation relies on the fact that a clique is a symmetric substructure, hence, it can be found by starting from any vertex it contains.

Fig. 1(b) shows how the graph in Fig. 1(a) is explored in the graph orientation approach to find all the 4-cliques. Assume that the edges are oriented in alphabetical order (from the earlier letter to the later letter). The first level contains all the vertices in the graph representing the root of their respective search trees. At the second level, each tree branches out from the root vertex to its neighbors. For example, the branch $A \rightarrow B$ indicates that there is an edge from vertex A to vertex B. At the third level, each edge branches out to the triangles it participates in. For example, the path $A \rightarrow B \rightarrow C$ indicates that there is a triangle containing vertices A, B, and C. Here, C is found by intersecting the adjacency lists of vertices A and B (i.e., $Adj(A) \cap Adj(B)$, where $Adj(v)$ is the adjacency list of a vertex v). Finally, at the fourth level, each triangle branches out to the 4-cliques it participates in. For example, the path $A \rightarrow B \rightarrow C \rightarrow D$ indicates that there is a 4-clique containing vertices A, B, C, and D. Here, D is found by intersecting the adjacency lists for A, B, and C. Since, $Adj(A) \cap Adj(B)$ was computed in the previous level, what remains is intersecting the previous result with $Adj(C)$. Since we are looking for 4-cliques, the tree traversal stops at the fourth level. In general, when looking for k-cliques, the tree traversal stops at level k.

Graph orientation has two main benefits. The first benefit is that it eliminates redundant clique discovery as previously mentioned. In the example in Fig. 1(b), the 4-clique containing vertices A, B, C, and D is only discovered in the tree rooted at vertex A. It is not redundantly discovered in the other trees because vertex A is not reachable from the other vertices in the directed graph. The second benefit of graph orientation is that it reduces the out-degrees of the vertices and the maximum out-degree of the graph. The maximum



(a) Example undirected graph

(b) Search tree for graph orientation approach

(c) Search tree for pivoting approach

(d) Search tree for pivoting approach with graph orientation approach used for level 1

(e) Search tree for pivoting approach with graph orientation approach used for levels 1 and 2

Figure 1: Counting 4-cliques in an example graph using different approaches

```
1  $numCliques = 0$
2  procedure $traverseSubtree(G, k, \ell, I)$
3      for $v \in I$
4          $I' = I \cap Adj_G(v)$
5          if $\ell + 1 == k$
6              $numCliques += |I'|$
7          else if $|I'| > 0$
8              $traverseSubtree(G, k, \ell + 1, I')$
```

Figure 2: Tree Traversal for Graph Orientation

out-degree has a quadratic impact on memory consumption, as we show in Section 2.4.

Fig. 2 shows the pseudocode for traversing a subtree in the graph orientation approach. The parameters are the graph G, the value of k, the current level ℓ, and a set of vertices I. I contains the intersection of the adjacency lists of all the vertices in the previous levels which is also the set of vertices at the current level representing the set of cliques at the current level. For example, if we are visiting level 3 along the path $A \longrightarrow B$, then ℓ will be 3 and I will be $Adj(A) \cap Adj(B)$ representing the common neighbors of A and B or the triangles that A and B are part of. The pseudocode iterates through the vertices at the current level (I) which represent the ℓ-cliques (line 3). For each vertex v, it intersects that vertex's adjacency list $Adj_G(v)$ with those of the previous levels' vertices (I) to find the vertices at the next level (I') which represent the $(\ell+1)$-cliques (line 4). If $\ell+1$ is k (line 5), then the total number of k-cliques is incremented by the number of k-cliques that were just found (line 6). Otherwise, if the set of $(\ell+1)$-cliques found is not empty (line 7), then they are visited at the next level (line 8).

Parallel k-clique counting based on graph orientation has been extensively studied on CPUs [11, 13, 40, 56]. Vertices or edges are typically distributed across CPU threads, and each thread performs a sequential depth-first traversal on the trees or subtrees corresponding to the vertices or edges assigned to the thread to count the underlying cliques. To the best of our knowledge, ARB-COUNT [56] is the best performing parallel CPU implementation of k-clique counting based on graph orientation. In this paper, we present our parallel GPU implementation of k-clique counting based on graph orientation and compare its performance with ARB-COUNT.

2.3 Pivoting Approach

Another approach to avoiding redundant clique discovery is pivoting [33]. The idea of pivoting is inspired by the Bron-Kerbosch maximal clique finding approach [18]. Pivoting relies on the fact that a $(k+i)$-clique consists of $\binom{k+i}{k}$ k-cliques, so instead of searching for all of these k-cliques, it is sufficient to find the largest $(k+i)$-clique and all the k-cliques it contains are found. For example, the graph in Fig. 1(a) contains a 5-clique consisting of vertices A, B, C, D, and J. This 5-clique contains $\binom{5}{4} = 5$ different 4-cliques. In the graph oriented approach in Fig. 1(b), these five 4-cliques are discovered by five different paths in the search trees. Instead, the pivoting approach just discovers the 5-clique and then concludes the existence of five 4-cliques.

Fig. 1(c) shows an example of how the graph in Fig. 1(a) is explored using the pivoting approach. As the search tree is traversed, at every branching point in the search tree, one pivot child vertex is selected which is typically the vertex that has the largest common number of neighbors with its parent. All the pivot's neighbors are then excluded while branching to the next level since these neighbors are eventually reachable from the pivot vertex (i.e., the pivot vertex is their parent in the search tree). For example, at the first level in Fig. 1(c), vertex B is selected as the pivot vertex because it has the largest number of neighbors. Accordingly, all of B's neighbors (A, C, D, E, J, K) are excluded from creating search trees at the first level. At the second level, when branching from vertex B to its neighbors, vertex D is selected as the pivot because it has the largest number of common neighbors with B. Accordingly, all of B and D's common neighbors (A, C, E, J, K) are excluded while branching to the second level. The tree traversal proceeds in this way. Unlike the graph orientation approach, the pivoting approach does not stop at level k because it is searching for the largest $(k+i)$-clique. It continues exploring until it reaches the bottom of the tree (or satisfies a stopping criteria [33]). To calculate the number of k-cliques found on a path, the combinatorial formula $\binom{n_p}{n_v - k}$ is used, where n_p is the number of pivots in the path, and n_v is the number of vertices in the path.

Fig. 3 shows the pseudocode for traversing a subtree in the pivoting approach. Compared to Fig. 2, it takes an additional parameter to track the number of pivot vertices encountered on the path. First,

```
1  $numCliques = 0$
2  procedure $traverseSubtree(G, k, \ell, I, nPivots)$
3      $v_{pivot} = findPivotVertex(I, G)$
4      $I_{pruned} = I - Adj_G(v_{pivot})$
5      for $v \in I_{pruned}$
6          $nPivots' = (v == v_{pivot})?(nPivots + 1) : nPivots$
7          if $\ell + 1 - k \leq nPivots'$
8              $I' = I \cap Adj_G(v) - \{u \in I_{pruned} | u < v\}$
9              if $|I'| > 0$
10                 $traverseSubtree(G, k, \ell + 1, I', nPivots')$
11             elseif $\ell + 1 \geq k$
12                 $numCliques += \binom{nPivots'}{\ell+1-k}$
```

Figure 3: Tree Traversal for Pivoting

the pivot vertex is found (line 3) and the neighbors of the pivot vertex are pruned from the level (line 4). Next, the code iterates through the remaining vertices which represent the ℓ-cliques (line 5). If the stopping criteria [33] has not been reached (line 7), then the vertex's adjacency list $Adj_G(v)$ is intersected with those of the previous levels' vertices (I) to find the vertices at the next level (I') which represent the $(\ell + 1)$-cliques (line 8). I' also excludes vertices at the current level that have already been visited to avoid finding redundant cliques (line 8). If I' is not empty (line 9) meaning that some $(\ell + 1)$-cliques are found, then the next level is visited (line 10). Otherwise, if there are no $(\ell + 1)$-cliques, then the current tree node represents the largest clique on this path. If the size of this large clique is $\geq k$ (line 11), then the total number of k-cliques is incremented by the number of k-cliques in the large clique just found (line 12).

Compared to graph orientation, pivoting has the advantage that it reduces the search space significantly by eliminating the neighbors of the pivot vertex from the branching. This reduction is clear when comparing Fig. 1(b) and Fig. 1(c). The reduction in branching makes pivoting particularly suitable for large k, or for counting all cliques for all k. On the other hand, pivoting has several disadvantages. The first disadvantage of pivoting is that it requires deeper exploration of the search tree which exacerbates load imbalance in parallel implementations. The second disadvantage of pivoting is that it reduces the amount of parallelism available by eliminating some of the search trees or subtrees and folding them into fewer and deeper trees. The third disadvantage of pivoting is that it requires an undirected graph, which makes adjacency list intersection operations prohibitively expensive for large graphs where vertices have very high degrees, hence very long adjacency lists. The second and third disadvantages are mitigated [33] by starting with the graph orientation approach for the first level of the tree using a directed graph, then switching to the pivoting approach for the remaining levels using an undirected induced sub-graph (see Section 2.4). Examples of this hybrid approach are illustrated in Fig. 1(d) and Fig. 1(e).

To our knowledge, Pivoter [33] is the only implementation of k-clique counting based on pivoting. The implementation is parallelized on the CPU by distributing vertices or edges across CPU threads, and having each thread perform a sequential depth-first traversal of the tree or subtree. In this paper, we present our parallel GPU implementation of k-clique counting based on pivoting and compare its performance to Pivoter.

2.4 Induced Sub-graph Optimization

As shown in Fig. 2 and Fig. 3, both the graph orientation and pivoting approaches spend a significant amount of time performing adjacency list intersection operations. Note that set difference can also be performed as an intersection operation because $A - B = A \cap \overline{B}$. A common optimization for the intersection operations is to shrink the size of the adjacency lists by removing from the graph, for each search tree, the vertices that will never be reached by the tree. For example, in Fig. 1, when traversing a tree rooted at the vertex A, only the neighbors of A can ever be reached. Hence, any vertex that is not a neighbor of A can be removed from the graph before the traversal.

In general, when traversing a search tree rooted at a vertex v, the first step is to extract the vertex-induced sub-graph consisting of the vertices in $Adj(v)$. This induced sub-graph is used throughout the tree traversal instead of the full graph. Since the induced sub-graph is typically much smaller than the full graph, it has smaller adjacency lists resulting in faster adjacency list intersection operations. Note that in principle, an induced sub-graph may be extracted at any level in the search tree. For example, if the tree contains a path $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_i$, the subtree rooted at v_i can only reach vertices that are neighbors of all the vertices $v_1, v_2, ..., v_i$. Therefore, the induced sub-graph consisting of the vertices in $Adj(v_1) \cap Adj(v_2) \cap ... \cap Adj(v_i)$ is sufficient for traversing the subtree. However, extracting the induced sub-graph at each level is usually not worth the overhead. The induced sub-graph is typically extracted at one level, either the first or the second. In this paper, both alternatives are explored.

The largest possible induced sub-graph has d_{max} vertices, where d_{max} is the maximum out-degree of the graph. Therefore, the upper bound on the size of an induced sub-graph is $O(d_{max}^2)$. Since the maximum out-degree of the graph has a quadratic impact on memory consumption, the choice of graph orientation is critical for reducing memory consumption, as mentioned in Section 2.2. It becomes even more critical in parallel implementations because when the trees or subtrees are processed in parallel and each has a different induced sub-graph, each needs its own memory space to store its induced sub-graph. Section 3.4 describes how the memory consumption of the induced sub-graphs is further reduced in our parallel GPU implementation.

3 PARALLEL CLIQUE COUNTING ON GPUS

3.1 Graph Format and Orientation Criteria

We represent the input graph using the hybrid Compressed Sparse Row (CSR) + Coordinate (COO) storage format. The CSR representation facilitates finding the adjacency list of a given vertex, which is useful for vertex-centric processing and parallelization. The COO representation facilitates finding the source and destination vertex of a given edge, which is useful for edge-centric processing and parallelization.

Before clique counting begins, we first orient the graph to become a directed graph. Recall that both the graph orientation approach and the pivoting approach require the graph to be oriented at the beginning. Our implementation supports two different orientation criteria: degree orientation and degeneracy orientation. Degree orientation orients edges from the vertex with the lower degree to the vertex with the higher degree. Degeneracy orientation orients edges from the lower k-core order to the higher k-core order. The k-core order is obtained from k-core decomposition which iteratively eliminates the minimum degree vertices from the graph. The k-core order is the order in which the vertex is removed from the graph.

We implement both orientation criteria on the GPU. In both cases, after determining which edges to keep, the undesired edges are filtered out and the CSR row pointers are recomputed with histogram and exclusive scan operations. The tradeoff between orientation criteria is evaluated in Section 4.4.

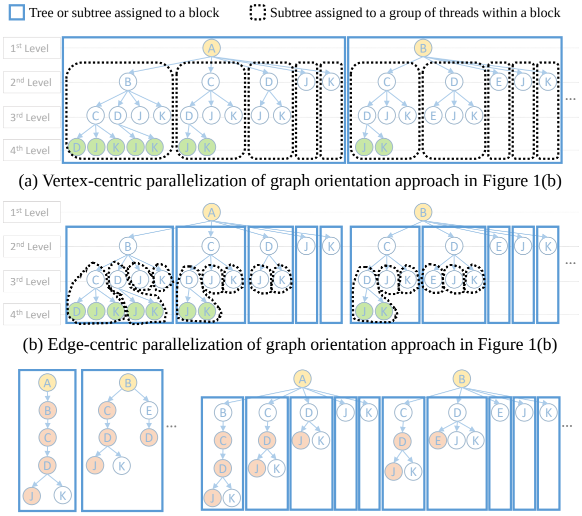
We note that although degree orientation and degeneracy orientation are currently supported, our implementation can easily be extended to support other orientation criteria. The choice of orientation criteria is orthogonal to our work and not intended as a contribution of this paper.

3.2 Parallelization Schemes

GPUs provide a massive amount of parallelism and are capable of running tens of thousands to hundreds of thousands of threads concurrently [32]. A grid of threads running on a GPU is divided into thread blocks. Threads in the same thread block can collaborate by synchronizing at a barrier and sharing a fast scratchpad memory (also called shared memory). Thread blocks are divided into warps which consist of 32 threads bound by the SIMD execution model. Threads in the same warp can collaborate using low cost warp-level primitives.

Our main strategy for parallelizing k-clique counting on GPUs is to traverse different search trees or subtrees in parallel. For both graph orientation and pivoting, we implement two different parallelization schemes: a vertex-centric scheme and an edge-centric scheme[1]. In the vertex-centric scheme, each thread block is assigned to a vertex of the input graph (level 1 in the search tree) and is responsible for traversing the tree rooted at that vertex. The threads in the block collaborate to extract the induced sub-graph consisting of the vertex's neighbors, then proceed to traverse the vertex's tree using the induced sub-graph. In the edge-centric scheme, each thread block is assigned to an edge of the input graph (level 2 in the search tree) and is responsible for traversing the subtree stemming from that edge. The threads in the block collaborate to extract the induced sub-graph consisting of the common neighbors of the edge's endpoints, then proceed to traverse the edge's subtree using the induced sub-graph.

The advantage of the vertex-centric scheme over the edge-centric scheme is that it extracts an induced sub-graph for each vertex's tree as opposed to each edge's subtree. Hence, the vertex-centric scheme amortizes the cost of extracting the induced sub-graph over the traversal of a larger tree. The advantage of the edge-centric scheme over the vertex-centric scheme is that it extracts more induced sub-graphs that are smaller in size. Hence, the edge-centric scheme exposes finer-grain parallelism which makes it more robust against load imbalance. It also results in shorter adjacency lists. We compare the performance of the vertex-centric and edge-centric schemes in Section 4.3.


Tree or subtree assigned to a block
Subtree assigned to a group of threads within a block
1st Level
2nd Level
3rd Level
4th Level
A
B
C
D
E
J
K
...

(a) Vertex-centric parallelization of graph orientation approach in Figure 1(b)

(b) Edge-centric parallelization of graph orientation approach in Figure 1(b)

(c) Vertex-centric parallelization of pivoting approach in Figure 1(d)

(d) Edge-centric parallelization of pivoting approach in Figure 1(e)

Figure 4: Parallelization schemes

Parallelization of work within each thread block differs between the graph orientation approach and the pivoting approach. In the graph orientation approach, we partition the blocks into groups of threads and each group independently traverses one of the subtrees in the next level. In the vertex-centric scheme, each group of threads is assigned to an outgoing edge (level 2 in the search tree) of the block's vertex, and the group independently traverses the subtree rooted at the edge. An example is shown in Fig. 4(a). In the edge-centric scheme, each group of threads is assigned to a triangle (level 3 in the search tree) that the block's edge participates in, and the group independently traverses the subtree rooted at that triangle. An example is shown in Fig. 4(b). In both cases, the threads in a group jointly perform a depth-first traversal of the subtree that the group is assigned to, visiting the nodes in the subtree sequentially. At each node of the subtree, the threads in the group collaborate to perform the adjacency list intersection operation in parallel. We discuss how we parallelize the adjacency list intersection operations in Section 3.4.

In the pivoting approach, we also partition the blocks into groups of threads, however these groups do not process next-level subtrees independently. Instead, all threads in the block stay together as they jointly perform a sequential depth-first traversal of the tree/subtree that the block is assigned to. However, at each node in the tree, identifying which neighbor is the pivot vertex (line 3 in Fig. 3) requires performing a list intersection operation for each of the neighbors. Therefore, each group of threads is assigned to a different neighbor and performs a list intersection operation for that neighbor to check if it is the pivot. Examples of the vertex-centric

[1] The terms *vertex-centric* and *edge-centric* in the context of parallel graph processing commonly refer to when different parallel workers are assigned to different vertices or edges, respectively. In some parts of the literature, they also imply that a parallel worker is restricted to accessing only its vertex's incident edges or its edge's endpoints, respectively. In our usage of the terms, we do not assume this restriction.

and edge-centric schemes for the pivoting approach are shown in Fig. 4(c) and Fig. 4(d), respectively.

There are two reasons why, in the pivoting approach, and unlike the graph orientation approach, groups of threads are not assigned to process the next-level subtrees independently. The first reason is that the process of finding the pivot element at each tree node is expensive and already provides enough work to be parallelized across groups. The second reason is that the pivoting approach has deeper trees than the graph orientation approach, thereby requiring more memory to store intermediate results. Parallelizing the next-level subtrees across groups of threads would require too much memory for storing the intermediate results of each subtree.

For now, a group of threads can be thought of as a warp, which is the most natural way to partition a block. However, we improve on this partitioning granularity in Section 3.5.

3.3 Traversing a Subtree

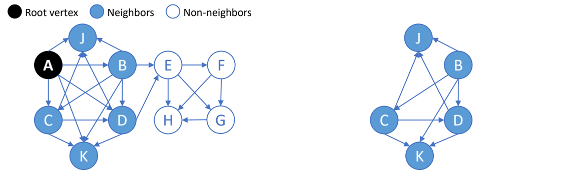
In Section 3.2, we saw how trees or subtrees are distributed across blocks or groups of threads to be traversed in parallel. Tree traversal on CPUs is typically done using recursion. However, using recursion on the GPU is not suitable because there is a large amount of intermediate traversal data that needs to be saved, and the tree is traversed jointly by multiple fine-grain threads that need to access the same intermediate traversal data. For this reason, our implementation replaces recursion with an iterative tree traversal whereby threads traversing the same tree explicitly manage a shared stack. We omit the details of how the recursive traversal is replaced with an iterative traversal due to space constraints.

The shared stack used in the iterative traversal is pre-allocated and provisioned for the maximum depth of the tree. The large components of each stack entry (such as vertex arrays) are preallocated in global memory, while the small components (such as counters) are pre-allocated in shared memory for fast access. A different stack is needed for each block or group that traverses a tree or subtree independently, which puts high pressure on the global memory capacity. To reduce this pressure, we employ various memory management techniques discussed in Section 3.6.

3.4 Binary Encoding of Induced Sub-graphs

Recall from Section 2.4 that the first step a thread block performs before traversing its tree or subtree is to extract an induced sub-graph. In the vertex-centric scheme, the induced sub-graph consists of the vertex's neighbors, whereas in the edge-centric scheme, the induced sub-graph consists of the common neighbors of the edge's endpoints. We use a directed induced sub-graph for the graph orientation approach, and an undirected induced sub-graph for the pivoting approach. The block's induced sub-graph is used by all threads in the block to perform the adjacency list intersection operations throughout the tree traversal.

The storage format used for the induced sub-graph is of utmost importance for both memory consumption and execution time. For memory consumption, since each thread block has a different induced sub-graph, enough memory must be allocated for all the blocks running simultaneously so they can each store a private induced sub-graph. Hence, the space efficiency of the induced sub-graph storage format is crucial for the overall memory consumption.



(a) Example graph (b) Induced sub-graph of neighbors of A

		Logical Adjacency List		Binary-Encoded Adjacency List	
		Directed	Undirected	Directed BCDJK	Undirected BCDJK
Source Vertex	B	{C, D, J, K}	{C, D, J, K}	01111	01111
	C	{D, J, K}	{B, D, J, K}	00111	10111
	D	{J, K}	{B, C, J, K}	00011	11011
	J	{}	{B, C, D}	00000	11100
	K	{}	{B, C, D}	00000	11100

(c) Logical and binary-encoded adjacency lists of induced sub-graph

Figure 5: Binary encoding example

For execution time, since threads use the induced sub-graph to perform adjacency list intersection operations, the storage format must be designed to enable low-latency parallel intersections.

To optimize both memory consumption and execution time, we use binary encoding to represent the induced sub-graph. To the best of our knowledge, our work is the first to use binary encoding for the induced sub-graphs in k-clique counting. Related work on the maximal clique problem uses binary encoding for the entire graph [59] or for specialized data structures to represent and operate on the candidate maximal cliques [67, 70]. Other graph processing works use binary encoding in different contexts.

Fig. 5 shows an example of how an induced sub-graph can be binary encoded. Assume that a thread block is assigned to traverse the tree rooted at vertex A in the graph in Fig. 5(a). The only vertices ever visited in that tree are the neighbors of A. Therefore, the thread block starts by extracting the induced sub-graph consisting of the neighbors of A shown in Fig. 5(b). This induced sub-graph is binary encoded as shown in Fig. 5(c). The adjacency list of each vertex in the sub-graph consists of a bit vector with a 1 for each neighbor and a 0 otherwise. In this example, the neighbor vertices are assigned to bit positions in alphabetical order. Any two lists can be intersected by performing a simple bitwise-and operation between the two bit vectors. Note that in addition to the induced sub-graph being binary encoded, all the intermediate vertex lists (i.e., I, I', and I_{pruned} in Fig. 2 and Fig. 3) are also binary encoded, and intersections with these intermediate vertex lists (line 4 in Fig. 2 and lines 4 and 8 in Fig. 3) also use bitwise-and.

The advantage of binary encoding for memory consumption is that each vertex in an adjacency list is represented with a single bit. Since dynamic memory allocation is not efficient on GPUs, the space for each thread block to store its induced sub-graph and its intermediate vertex lists must be pre-allocated with enough capacity for the largest possible sub-graph. The largest possible sub-graph may have d_{max} vertices, where d_{max} is the maximum out-degree of the graph, and the sub-graph may be completely dense. In this case, storing the sub-graph would require $O(d_{max}^2)$ memory, and storing the intermediate vertex lists would require $O(d_{max})$ memory per level. We show in Section 4.4 that with a proper orientation criterion, the value of d_{max} remains manageable even for very large graphs. Nevetheless, binary encoding has the advantage of reducing the amount of memory needed for storing the sub-graph and the intermediate vertex lists by a factor of 32.

The advantage of binary encoding for execution time is that it allows list intersection operations to be performed using simple bitwise-and operations. Traditional adjacency list intersection techniques on GPUs are complex to parallelize, suffer from control divergence, and exhibit uncoalesced memory access patterns. On the other hand, performing bitwise-and on a bit vector is easy to parallelize across threads in a group or block, does not suffer from control divergence, and enables coalescing of memory accesses.

The reason binary encoding is particularly effective for the induced sub-graphs and intermediate vertex lists is that they typically consist of a small number of vertices, especially when a good graph orientation criteria is used. Moreover, the induced sub-graphs are typically denser than the full graph. In contrast, binary encoding for the full graph is impractical because the full graph has many more vertices and is usually much more sparse, resulting in many wasted 0 bits. For this reason, we continue to represent the full graph using the hybrid CSR+COO format (see Section 3.1) and only represent the induced sub-graphs using binary encoding. To extract the binary encoded induced sub-graph from the full hybrid CSR+COO graph, we intersect the adjacency lists of the hybrid CSR+COO graph using binary-search-based intersections.

We evaluate the performance improvement of using binary encoded induced sub-graphs in Section 4.3.

3.5 Sub-warp Partitioning

In Section 3.2, we explained that for both the graph orientation approach and the pivoting approach, our implementation partitions thread blocks into groups of threads and distributes the block's work across these groups. In the graph orientation approach, a block is assigned to a tree or subtree, and has each group of threads process one of the next level subtrees in parallel. In the pivoting approach, a block is also assigned to a tree or subtree, but the groups of threads jointly iterate over the tree nodes sequentially. However, for each tree node, when determining which child vertex is the pivot, each group of threads is used to check a different child in parallel.

One important design consideration is the granularity at which thread blocks are partitioned into groups. The most natural granularity is the warp because threads in the same warp are bound together by SIMD and are able to collaborate using low cost warp-level primitives. However, the introduction of binary encoding makes the warp granularity often too coarse. Recall that threads within a group collaborate to perform a single list intersection operation in parallel. With binary encoding, each thread can intersect 32 list elements simultaneously with a single bitwise-and operation. Therefore, to fully utilize all 32 threads in the warp, the intersection needs to contain 1024 list elements. We show in Section 4.4 that with proper graph orientation criteria, the maximum out-degree (and consequently, the largest binary encoded list size) is often much smaller than that. Therefore, partitioning blocks at the warp granularity would lead to underutilization of parallel execution resources.

To address this issue, we implement sub-warp partitioning where thread blocks are partitioned to groups smaller than a warp. Since the NVIDIA Volta architecture, the *independent thread scheduling* has enabled fine-grain collaboration between a subset of threads within a warp. We leverage this feature to enable the creation of thread groups that are 32, 16, 8, 4, 2, or 1 threads in size. The number of threads per group is a tunable parameter and the same traversal code works for any group size. We evaluate the performance improvement of using sub-warp partitioning in Section 4.3.

3.6 Memory Management

The issue of memory consumption is exacerbated on the GPU compared to the CPU for two key reasons. The first reason is that the capacity of the device memory on a typical GPU is much smaller than the capacity of main memory on a typical CPU. The second reason is that GPUs are massively parallel processors so they perform much more work in parallel, thereby requiring much more intermediate data to be stored simultaneously. For example, a CPU may run tens of threads at a time, so it only needs enough memory to maintain that many different sets of induced sub-graphs and intermediate vertex lists. In contrast, a GPU may run hundreds to thousands of thread blocks at once, so it needs enough memory to maintain hundreds to thousands of different sets of induced sub-graphs and intermediate vertex lists. Therefore, with a lower memory capacity and a higher demand for memory, it becomes critical to manage memory efficiently on the GPU.

We have discussed multiple techniques that we use for reducing memory consumption throughout this paper. In Section 3.2, we discuss how an induced sub-graph is extracted once per thread block and shared by multiple groups of threads. In Section 3.4, we discuss how binary encoding of sub-graphs and intermediate vertex lists reduces their memory requirement. In this section, we discuss one more technique for reducing memory consumption.

Recall that memory needs to be pre-allocated for each block to store the induced sub-graph and intermediate vertex lists that it uses. If we assign one vertex or edge to each block, we will launch many more blocks than the number that can execute simultaneously, which means that the pre-allocated memory spaces will not always be utilized. To mitigate this inefficiency, we instead launch the maximum number of thread blocks that can run simultaneously and reuse these thread blocks to process multiple vertices or edges (by incrementing a global counter). In doing so, we reduce the number of pre-allocated memory spaces and reuse the same memory space to process multiple vertices or edges.

4 EVALUATION

4.1 Methodology

Evaluation Platform. In this section, we evaluate our GPU implementations on an NVIDIA Volta V100 GPU with 32GB of memory. The GPU is attached to an Intel Xeon Gold 6230 CPU. We use a single CPU thread to drive the GPU. For a broader evaluation, we also report the execution times of our GPU implementations on an NVIDIA Ampere A100 GPU and an NVIDIA Ampere RTX 3090 GPU in Table 3 at the end of this paper.

Datasets. We use the same graphs used by ARB-COUNT [56] for exact k-clique counting evaluation. The details of these graphs are shown in Table 1. These graphs are real-world undirected graphs from the Stanford Network Analysis Project (SNAP) [38].

Table 1: Execution time and memory consumption of our GPU implementations

					Execution Time (s)				Memory Consumption (MB)	
Graph	|V|	|E|	k	# k-Cliques	ARB-COUNT [56]	Pivoter [33]	GPU-Graph Orientation	GPU-Pivot	GPU-Graph Orientation	GPU-Pivot
as-skitter	1,696,415	11,095,298	4	148,834,439	0.60	20.90	0.034	0.459	152.09	149.21
			5	1,183,885,507	0.67	22.54	0.069	0.721	154.59	
			6	9,759,000,981	1.24	25.30	0.245	1.01	157.09	
			7	73,142,566,591	5.73	27.14	1.434	1.269	152.09	
			8	481,576,204,696	28.38	27.46	9.531	1.585	162.09	
			9	2,781,731,674,867	158.45	28.45	68.221	1.835	164.59	
			10	14,217,188,170,569	854.87	28.45	474.785	1.777	167.09	
			11	64,975,151,572,336	4,158.53	28.45	2,997.385	1.777	169.59	
com-dblp	317,080	1,049,866	4	16,713,192	0.10	2.88	0.008	0.109	26.64	23.38
			5	262,663,639	0.13	2.88	0.016	0.109	29.14	
			6	4,221,802,226	0.30	2.88	0.042	0.109	31.64	
			7	60,913,718,813	2.05	2.88	0.545	0.109	26.64	
			8	777,232,734,905	24.06	2.88	9.031	0.109	36.64	
			9	8,813,264,533,265	281.39	2.88	139.046	0.109	39.14	
			10	89,563,892,212,629	2,981.74	2.88	2,262.99	0.109	41.64	
			11	822,551,101,011,469	>5 hours	2.88	>5 hours	0.109	44.14	
com-orkut	3,072,441	117,185,083	4	3,221,946,137	3.10	292.35	0.426	8.83	1,394.40	1,400.83
			5	15,766,607,860	4.94	385.04	1.014	13.869	1,399.40	
			6	75,249,427,585	12.57	462.05	3.506	17.229	1,404.40	
			7	353,962,921,685	42.09	517.29	11.719	20.331	1,394.40	
			8	1,632,691,821,296	150.87	559.75	45.319	26.137	1,414.40	
			9	7,248,102,160,867	584.39	598.88	212.912	33.644	1,419.40	
			10	30,288,138,110,629	2,315.89	647.18	1,002.165	39.957	1,424.40	
			11	117,138,620,358,191	8,843.51	647.18	4,421.597	48.101	1,429.40	
com-friendster	65,608,366	1,806,067,135	4	8,963,503,263	109.46	Out of memory	10.215	44.897	21,209.19	21,220.95
			5	21,710,817,218	111.75	Out of memory	11.796	53.874	21,215.44	
			6	59,926,510,355	115.52	Out of memory	17.22	63.874	21,221.69	
			7	296,858,496,789	139.98	Out of memory	45.697	66.544	21,209.19	
			8	3,120,447,373,827	300.62	Out of memory	99.866	67.064	21,234.19	
			9	40,033,489,612,826	1,796.12	Out of memory	803.53	71.404	21,240.44	
			10	487,090,833,092,739	16,836.41	Out of memory	12,775.67	71.051	21,246.69	
			11	5,403,375,502,221,430	>5hours	Out of memory	>5hours	71.448	21,252.94	
com-lj	3,997,962	34,681,189	4	5,216,918,441	1.77	268.06	0.104	10.864	513.39	499.47
			5	246,378,629,120	7.52	1,475.99	0.943	68.966	524.02	
			6	10,990,740,312,954	258.46	7,816.13	23.792	379.88	534.64	
			7	449,022,426,169,164	10,733.21	>5 hours	1,077.66	1,639.537	513.39	
			8	16,890,998,195,437,600	>5hours	>5 hours	>5 hours	6,850.989	555.89	

Baselines. We compare our GPU implementations to two CPU baselines: ARB-COUNT [56] and Pivoter [33]. ARB-COUNT [56] represents the state-of-the-art parallel graph orientation implementation on CPU, which significantly outperforms other graph orientation implementations [13, 40]. Pivoter [33] represents the state-of-the-art parallel pivoting implementation on CPU. In this section, we use the execution times reported by ARB-COUNT [56] for the parallel implementations of ARB-COUNT and Pivoter. These times are obtained using an Intel Xeon Scalable (Cascade Lake) processor with 30 cores (60 threads) and 240 GB of main memory. For a broader evaluation, we also compare the execution times of our GPU implementations to the execution times reported by Lonkar and Beamer [42] in Table 3 at the end of this paper. These times are obtained using an Intel Xeon Platinum 8260 processor with 48 cores (96 threads) and 768 GB of main memory, but are only reported for up to $k = 8$.

Reporting of Measurements. The execution times we report include the time spent pre-processing and counting, and exclude the time spent reading the graph from disk. Unless otherwise specified, we report the time achieved for the best combination of orientation criteria (degree or degeneracy), parallelization scheme (vertex-centric or edge-centric), and sub-warp partition size. However, we make suggestions for how to select a good combination of these parameters in Section 4.5. Similar to ARB-COUNT [56], we do not report times greater than five hours.

4.2 Comparison with Parallel CPU Implementations

Fig. 6 compares the execution time of the state-of-the-art parallel CPU implementations with our GPU implementations for both the graph orientation approach and the pivoting approach. These execution times are also reported in Table 1, along with details about each graph and the memory consumed by each of our implementations for each graph. The missing datapoints in the Fig. 6 are those that take longer than 5 hrs to execute or run out of memory, as shown in Table 1. Note that our GPU implementations do not run out of memory for any scenario, despite the constrained GPU memory capacity. Based on the results in Fig. 6 and Table 1, we make two key observations.

Graph Orientation vs. Pivoting. The first observation is that the graph orientation approach performs better than the pivoting approach for small values of k, while the latter performs better for large values of k. This observation is consistent with that made in prior work [56]. The observation applies for both CPU and GPU implementations. Recall from Section 2.3 that pivoting has the advantage of reducing the amount of branching but the disadvantage

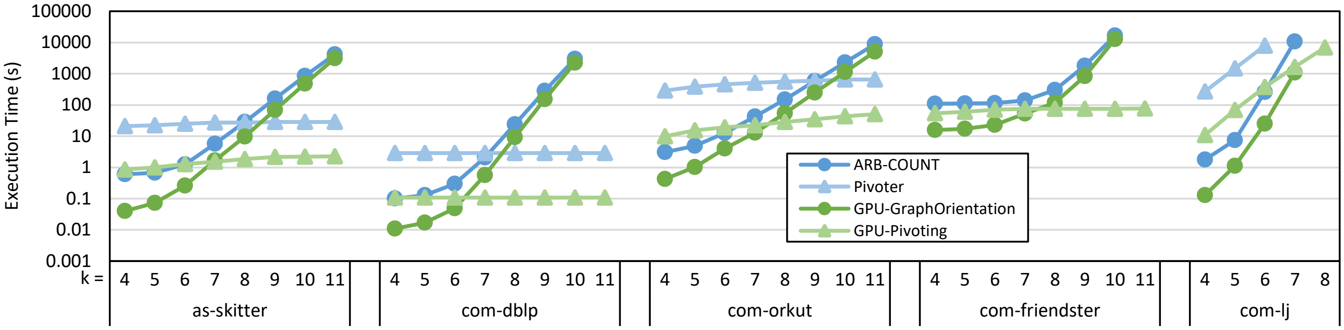
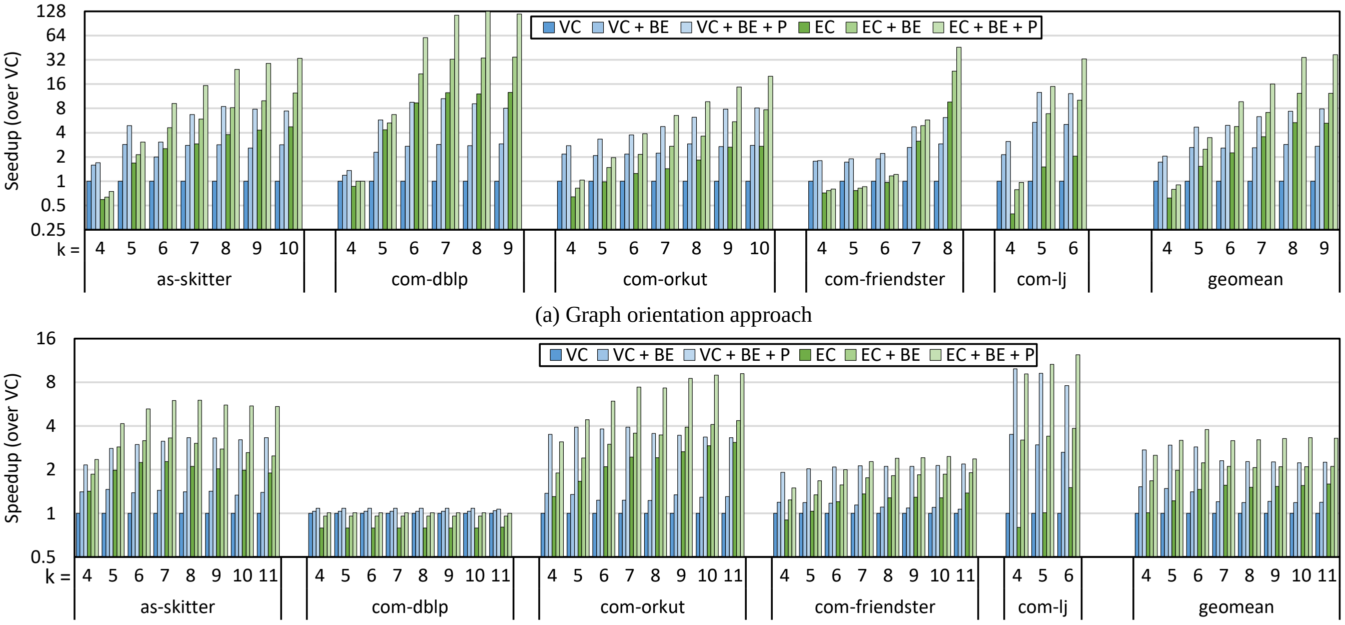


Figure 6: Comparing execution time with state-of-the-art parallel CPU implementations (lower is better)



(a) Graph orientation approach

(b) Pivoting approach

Figure 7: Impact of parallelization schemes and speedup of optimizations for each approach (higher is better)

of having deeper search trees. For small values of k, branching for the graph orientation approach is moderate, whereas the deep trees in the pivoting approach create load imbalance. However, as k gets larger, the branching drastically increases causing the graph orientation approach to suffer. In most cases, the transition from the graph orientation approach being fastest to the pivoting approach being fastest happens at around $k = 7$.

GPU vs. CPU. The second observation is that our best GPU implementation consistently and significantly outperforms the best parallel state-of-the-art CPU implementation across all values of k. Our best GPU implementation outperforms the best CPU implementation by a geometric mean speedup of 12.39×, 6.21×, and 18.99× for $k = 4$, 7, and 10, respectively. This result demonstrates the effectiveness of GPUs at accelerating k-clique counting.

4.3 Impact of Parallelization Schemes and Optimizations

Fig. 7(a) and Fig. 7(b) show the incremental speedup of binary encoding and sub-warp partitioning for both parallelization schemes for the graph orientation approach and the pivoting approach, respectively. VC and EC refer to the vertex-centric and edge-centric parallelization schemes, respectively, with induced sub-graphs represented using the CSR format, parallel list intersections performed using the binary search approach, and blocks partitioned into groups at warp granularity. The +BE suffix refers to when binary encoding is applied to the induced sub-graphs instead of using CSR and parallel list intersections are performed using bitwise-and operations. The +P suffix refers when sub-warp partitioning is applied and the best partition size is used. The omitted datapoints in the figure are those where the baseline (VC) takes longer than 5 hrs to run.

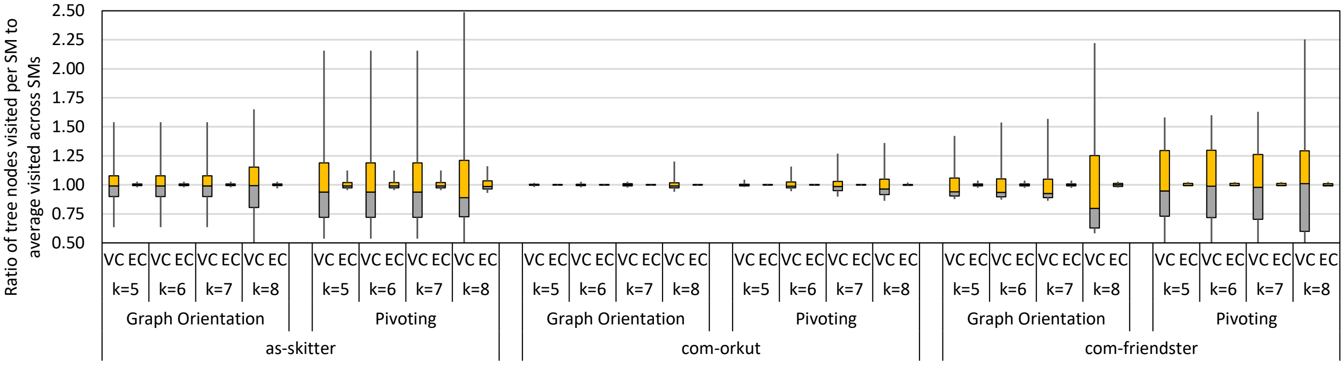


Figure 8: Load imbalance for a select set of graphs

Parallelization Scheme. We observe from Fig. 7 that the vertex-centric parallelization scheme is more effective than the edge-centric parallelization scheme for the initial values of k, particularly for the graph orientation approach. For example, for $k = 4$, VC+BE+P has a geometric mean speedup over EC+BE+P of 2.27× for the graph orientation approach and 1.09× for the pivoting approach. However, as k gets larger, the edge-centric parallelization scheme becomes more effective. For example, for $k = 9$, EC+BE+P has a geometric mean speedup over VC+BE+P of 4.68× for the graph orientation approach and 1.45× for the pivoting approach. Recall from Section 3.2 that the vertex-centric scheme has the advantage of amortizing the cost of extracting the induced sub-graph over a larger tree, whereas the edge-centric scheme has the advantage of extracting more parallelism making it more robust against load imbalance. The graph orientation approach for small k has the smallest trees, hence amortizing the induced sub-graph extraction across larger trees makes the vertex-centric scheme attractive. However, as k increases, load imbalance increases, which makes the edge-centric scheme attractive. We also note that the transition from the vertex-centric scheme being fastest to the edge-centric scheme being fastest happens at a much smaller k for our GPU implementation than it does for prior parallel CPU implementations [56]. GPUs exhibit more parallel execution resources than CPUs which makes them more sensitive to load imbalance, thereby favoring an earlier transition to the more load-balanced edge-centric scheme. Overall, we observe that selecting the vertex-centric scheme for $k < 6$ and the edge-centric scheme for $k \geq 6$ gives the best or near-best result in most cases.

To further study the impact of the edge-centric scheme on load imbalance, Fig. 8 shows the distribution of work across SMs for a select set of graphs for different approaches and values of k. The load of an SM is measured as the number of tree nodes visited by the SM normalized to the average number of tree nodes visited by all SMs. As expected, the load imbalance is higher for pivoting than for graph orientation because pivoting has fewer and deeper search trees. The load imbalance also increases with the value of k because the search trees become deeper. Most notably, we observe that the edge-centric scheme consistently has better load balance than the vertex-centric scheme.

For com-dblp with pivoting, we observe from Fig. 7(b) that there is little performance impact from the choice of parallelization scheme, not to mention other optimizations. The reason is that com-dblp has many clusters (it has the highest clustering coefficient [38] among the graphs), which makes the pivoting optimization particularly effective on it, leaving little room for other optimizations to have a significant impact.

Binary Encoding. We observe from Fig. 7 that binary encoding gives consistent performance improvement for both the graph orientation and pivoting approaches across all graphs, parallelization schemes, and values of k. The geometric mean speedup of applying binary encoding is 2.17× for the graph orientation approach, and 1.38× for the pivoting approach. Recall from Section 3.4 that binary encoding improves execution time because it enables lower-latency parallel list intersection operations. The graph orientation approach spends the majority of time performing list intersection operations, whereas the pivoting approach performs other kinds of operations like finding a maximum. For this reason, it is expected that the speedup of binary encoding would be more pronounced in the graph orientation approach.

Sub-warp Partitioning. We observe from Fig. 7 that sub-warp partitioning also gives consistent performance improvement for both the graph orientation and pivoting approaches across all graphs, parallelization schemes, and values of k. The geometric mean speedup of applying sub-warp partitioning with the best partition size is 1.98× for the graph orientation approach, and 1.73× for the pivoting approach. Recall from Section 3.2 that the groups of threads within a block in the graph orientation approach operate completely independently, whereas in pivoting, these groups collaborate with each other to find the pivot vertex. For this reason, it is expected that the speedup of unleashing more parallelism via sub-warp partitioning will be more pronounced in the graph orientation approach.

Regarding the choice of the best partition size, from our experience, partition sizes of 32 and 16 are never favorable. Aside from these, selecting the wrong partition size results in a geometric mean reduction in speedup of 1.24× in the average case and 1.65× in the worst case, which is within the speedup margin of sub-warp partitioning. Hence, sub-warp partitioning is still beneficial even if the best partition size is not correctly selected. In addition, we

Table 2: Impact of graph orientation criteria

	Undirected	Degree Orientation		Degeneracy Orientation	
Graph	d_{max}	d_{max}	Preprocessing Time in seconds	d_{max}	Preprocessing Time in seconds
as-skitter	35,455	231	0.005	111	0.205
com-dblp	343	113	0.002	113	0.051
com-orkut	33,313	535	0.056	253	0.833
com-friendster	5,214	868	5.465	304	12.294
com-lj	14,815	524	0.016	360	0.421

have found that for graph orientation, graphs with lower maximum degree (i.e., < 200) favor having fewer threads per group (e.g., 1 or 2), whereas graphs with higher maximum degree (i.e., ≥ 200) favor having more threads per group (e.g., 8). Graphs with higher maximum degree have larger intermediate adjacency lists that need to be interstected, so more threads can be utilized in performing the intersection operations in parallel. For pivoting, we have found that having fewer threads per group (e.g., 1 or 2) gives the best performance in most cases.

4.4 Impact of Graph Orientation Criteria

Table 2 shows the achieved maximum out-degree of the graph and the pre-processing time of the two different orientation criteria used in this work, namely degree orientation and degeneracy orientation. It is clear that degeneracy orientation achieves lower maximum out-degree but has higher pre-processing time. Recall that the maximum out-degree forms an upper bound on the length of the adjacency list intersections. Therefore, a lower maximum out-degree results in faster list intersection operations throughout the traversal.

Our analysis of the best orientation criteria for different runs shows that when the graph orientation approach is used, runs with lower values of k (i.e., $k < 7$) favor degree orientation, whereas runs with higher values of k favor degeneracy orientation. Runs with higher values of k take longer and perform more list intersections, hence there is enough work reduction to amortize degeneracy orientation's higher pre-processing cost. However, runs with lower values of k do not perform enough list intersections to amortize the pre-processing cost. On the other hand, when pivoting is used, degeneracy orientation is always preferred. Degeneracy orientation maximizes the effectiveness of pivoting at eliminating branches of the search tree.

The trends observed in this subsection are consistent with the trends observed in prior work [56]. We include this analysis here for completeness, however, as mentioned in Section 3.1, the choice of orientation criteria is orthogonal to our work and not intended as a contribution.

4.5 Algorithm and Parameter Selection

Throughout this section, unless otherwise specified, we have reported results for the best choice of algorithm and optimization parameters to show the maximum potential of our approach. However, when users solve for a particular graph and value of k, they face the challenge of selecting the best algorithm and optimization parameters. In this subsection, we make recommendations for how to make this selection based on our empirical analysis.

There are four selections that need to be made in our approach: (1) the algorithm (graph orientation or pivoting), (2) the orientation criteria (degree or degeneracy), (3) the parallelization scheme (vertex-centric or edge-centric), and (4) the sub-warp partition size. For selecting the algorithm, we observe in Section 4.2 that selecting graph orientation for $k < 7$ and pivoting for $k \geq 7$ gives the best result in most cases. For selecting the orientation criteria, we observe in Section 4.4 that graph orientation favors degree orientation when $k < 7$ and degeneracy orientation otherwise, whereas pivoting always favors degeneracy orientation. For selecting the parallelization scheme, we observe in Section 4.3 that selecting the vertex-centric scheme for $k < 6$ and the edge-centric scheme for $k \geq 6$ gives the best result in most cases. For selecting the sub-warp partition size, we observe in Section 4.3 that for graph orientation, graphs with maximum degree < 200 favor partition sizes of 1 or 2, and graphs with maximum degree ≥ 200 favor a partition size of 8, whereas for pivoting, a partition size of 1 is usually favorable.

Following these guidelines, users can select a near-optimal combination of algorithm and optimization parameters in the majority of cases. Compared to when the best combination is selected every time, the execution times achieved if these guidelines are followed are only 1.17× slower (geometric mean), which is well within the margin of speedups reported in this paper.

5 RELATED WORK

Graph Orientation Approach to Clique Counting. Graph orientation is a fundamental approach to avoiding redundant clique discovery [11, 13, 20, 21, 40, 42, 56]. To our knowledge, ARB-COUNT [56] is the state-of-the-art parallel implementation of k-clique counting on CPUs based on graph orientation. We implement the graph orientation approach for k-clique counting on the GPU and compare our performance with ARB-COUNT [56].

Pivoting Approach to Clique Counting. Pivoter [33] is a recent work on k-clique counting that is inspired by the classical pivoting idea of Bron-Kerbosh's maximal clique finding [18]. ARB-COUNT [56] compares to Pivoter and shows that Pivoter is advantageous for large k values. Pivoter is implemented on the CPU. We implement the pivoting approach for k-clique counting on the GPU and compare our performance with Pivoter [33].

Maximal Clique Enumeration. Enumerating the maximal cliques in a graph has been extensively studied on CPUs [10, 14, 55, 68] and GPUs [30, 34, 39, 59, 67, 70]. The pivoting approach is inspired by techniques used in maximal clique enumeration. Our work solves the k-clique counting problem, but our techniques can be extended to the maximal clique problem. To the best of our knowledge, none of the GPU works on maximal clique use edge-centric parallelization, use binary encoding for the induced sub-graph, or use sub-warp partitioning.

Triangle Counting. Many works perform triangle counting on the CPU [2, 29, 36, 46] or the GPU [4, 5, 25, 26, 31, 44, 47, 49, 62, 65]. A triangle is a 3-clique which is a special case of a k-clique. Our implementation performs k-clique counting for any k value.

Generalized Sub-graph Matching. Many works perform generalized sub-graph matching on the CPU [1, 17, 50, 54, 66] and the GPU [9, 16, 28, 41, 57, 63, 64, 73]. These frameworks search for an arbitrary k-vertex sub-graph and support different values of k. Due to generalization, such sub-graph matching frameworks suffer from memory explosion or prolonged execution times. Our implementation is specialized for k-cliques which are an important

Table 3: A comparison of the total execution time (in seconds) between ARB-COUNT [56] and Pivoter [33] on two different CPUs, and our GPU implementations on three different GPUs

		Intel Xeon Scalable (60 Threads)		Intel Xeon Platinum 8260 (96 Threads)		Nvidia Volta V100		Nvidia Ampere RTX3090		Nvidia Ampere A100	
Graph	k	ARB-COUNT	Pivoter	ARB-COUNT	Pivoter	GPU-Graph Orientation	GPU-Pivot	GPU-Graph Orientation	GPU-Pivot	GPU-Graph Orientation	GPU-Pivot
as-skitter	4	0.60	20.90	0.073	32.667	0.034	0.459	0.026	0.525	0.028	0.546
	5	0.67	22.54	0.127	35.333	0.069	0.721	0.039	0.654	0.034	0.682
	6	1.24	25.30	0.635	37.000	0.245	1.01	0.172	0.896	0.150	0.962
	7	5.73	27.14	5.28	39.000	1.434	1.269	0.886	1.135	0.729	1.314
	8	28.38	27.46	39.699	39.667	9.531	1.585	7.578	1.398	5.804	1.731
	9	158.45	28.45	-	-	68.221	1.835	59.026	1.634	46.297	1.967
	10	854.87	28.45	-	-	474.785	1.777	414.812	1.694	351.133	2.217
	11	4,158.53	28.45	-	-	2,997.385	1.777	2,572.444	1.764	2,493.202	2.15
com-dblp	4	0.10	2.88	-	-	0.008	0.109	0.009	0.118	0.010	0.112
	5	0.13	2.88	-	-	0.016	0.109	0.016	0.118	0.015	0.112
	6	0.30	2.88	-	-	0.042	0.109	0.031	0.118	0.030	0.112
	7	2.05	2.88	-	-	0.545	0.109	0.408	0.118	0.444	0.112
	8	24.06	2.88	-	-	9.031	0.109	7.912	0.118	8.835	0.112
	9	281.39	2.88	-	-	139.046	0.109	150.483	0.118	159.794	0.112
	10	2,981.74	2.88	-	-	2,262.99	0.109	2,190.995	0.118	2,435.950	0.112
	11	>5 hours	2.88	-	-	>5 hours	0.109	>5 hours	0.118	>5 hours	0.112
com-orkut	4	3.10	292.35	1.614	308.667	0.426	8.83	0.328	8.534	0.323	6.612
	5	4.94	385.04	2.863	400.667	1.014	13.869	0.733	14.403	0.606	10.926
	6	12.57	462.05	8.694	481.333	3.506	17.229	2.344	19.244	1.827	14.046
	7	42.09	517.29	33.278	525.333	11.719	20.331	8.522	22.918	7.681	17.321
	8	150.87	559.75	133.695	583.333	45.319	26.137	39.968	28.159	30.002	22.483
	9	584.39	598.88	-	-	212.912	33.644	193.653	34.694	140.732	29.82
	10	2,315.89	647.18	-	-	1,002.165	39.957	925.879	41.393	668.427	38.218
	11	8,843.51	647.18	-	-	4,421.597	48.101	4,086.51	46.564	3,150.510	44.815
com-friendster	4	109.46	Out of memory	70.01	4,433.5	10.215	44.897	Out of memory	Out of memory	9.126	33.202
	5	111.75	Out of memory	70.817	4,489.5	11.796	53.874	Out of memory	Out of memory	10.356	38.212
	6	115.52	Out of memory	74.418	4,554.5	17.22	63.874	Out of memory	Out of memory	14.391	47.318
	7	139.98	Out of memory	93.549	4,537.5	45.697	66.544	Out of memory	Out of memory	31.034	47.408
	8	300.62	Out of memory	385.024	4,556.5	99.866	67.064	Out of memory	Out of memory	74.857	47.07
	9	1,796.12	Out of memory	-	-	803.53	71.404	Out of memory	Out of memory	660.689	46.115
	10	16,836.41	Out of memory	-	-	12,775.67	71.051	Out of memory	Out of memory	11,911.473	45.224
	11	>5hours	Out of memory	-	-	>5hours	71.448	Out of memory	Out of memory	>5 hours	44.311
com-lj	4	1.77	268.06	0.416	299.500	0.104	10.864	0.09	9.339	0.093	8.529
	5	7.52	1,475.99	5.587	1,500.000	0.943	68.966	0.725	72.087	0.695	53.79
	6	258.46	7,816.13	256.241	>1 hour	23.792	379.88	23.15	404.725	18.697	301.772
	7	10,733.21	>5 hours	>1 hour	>1 hour	1,077.66	1,639.537	1,400.245	1,836.612	952.845	1,396.37
	8	>5hours	>5 hours	>1 hour	>1 hour	>5 hours	6,850.989	>5 hours	7,065.479	>5 hours	5,467.176

special case of a *k*-vertex sub-graph. Specializing for *k*-cliques enables optimizations that are not applicable to general sub-graphs, thereby providing better scalability for large values of *k*.

Truss Decomposition. Recently, *k*-truss decomposition has received significant attention on CPUs [12, 43, 48, 61] and GPUs [3, 6–8, 15, 24]. A *k*-truss is a relaxation of a *k*-clique. Our work solves the *k*-clique counting problem.

List Intersections. List intersection is an important operation at the heart of many sub-graph search algorithms. Different list intersection strategies have been proposed for GPUs such as pointer-chasing [3], binary-search [49], merge-path [26], hash-based [47], tile-based [15], bitmap-based [5], and others. Our implementations use binary-search intersections for extracting induced sub-graphs, and binary encoding for the lists in the induced sub-graph.

Parallel Search Tree Traversal on GPUs. Parallel search tree traversal on GPUs has been studied for other graph problems such as minimum vertex cover [69]. This work uses a global worklist [35] for dynamic load balancing because the search tree is narrow and highly imbalanced. Our work shows that for the *k*-clique counting problem, edge-centric parallelization is sufficient for achieving reasonable load balance.

6 CONCLUSION

We present parallel GPU implementations of *k*-clique counting that support both the graph orientation and pivoting approaches for eliminating redundant clique discovery. We explore vertex-centric and edge-centric parallelization schemes and apply various optimizations such as binary encoding and sub-warp partitioning to reduce memory consumption and efficiently utilize parallel execution resources. To the best of our knowledge, our work is the first GPU solution specialized for *k*-clique counting.

Our evaluation shows that our best GPU implementation substantially outperforms the best state-of-the-art parallel CPU implementation. Our efficient memory management strategies enable us to process very large graphs with billions of edges for arbitrary values of *k*. We also analyze the trade-offs between parallelization schemes, and show that the binary encoding and sub-warp partitioning optimizations yield significant performance gains.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous reviewers for their constructive feedback during the review process. This work was supported in part by the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR).

REFERENCES

[1] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, Nick G Duffield, and Theodore L Willke. 2017. Graphlet decomposition: Framework, algorithms, and applications. *Knowledge and Information Systems* 50, 3 (2017), 689–722.

[2] Mohammad Al Hasan and Vachik S Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 2 (2018), e1226.

[3] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on k-truss decomposition on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[4] Mohammad Almasri, Neo Vasudeva, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2021. HyKernel: A Hybrid Selection of One/Two-Phase Kernels for Triangle Counting on GPUs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[5] Mauro Bisson and Massimiliano Fatica. 2018. Update on static graph challenge on GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–8.

[6] M. Bisson and M. Fatica. 2018. Update on Static Graph Challenge on GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC.2018.8547514

[7] M. Blanco, T. M. Low, and K. Kim. 2019. Exploration of Fine-Grained Parallelism for Load Balancing Eager K-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916473

[8] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating truss decomposition on heterogeneous processors. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1751–1764.

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1190–1205.

[10] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1240–1248.

[11] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[12] A. Conte, D. De Sensi, R. Grossi, A. Marino, and L. Versari. 2018. Discovering k-Trusses in Large-Scale Networks. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547735

[13] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[14] Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura. 2020. Shared-memory parallel maximal clique enumeration from static and dynamic graphs. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–28.

[15] Safaa Diab, Mhd Ghaith Olabi, and Izzat El Hajj. 2020. KTrussExplorer: Exploring the Design Space of K-truss Decomposition Optimizations on GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[16] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun E Xiong, and Wen-mei Hwu. 2022. PARSEC: PARallel Subgraph Enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[17] Ethan R Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G Dimakis. 2016. Distributed estimation of graph 4-profiles. In *Proceedings of the 25th International Conference on World Wide Web*. 483–493.

[18] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation*. Springer, 403–414.

[19] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1719–1732.

[20] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. 2015. Clique counting in MapReduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)* 20 (2015), 1–20.

[21] Lukas Gianinazzi, Maciej Besta, Yannick Schaffner, and Torsten Hoefler. 2021. Parallel Algorithms for Finding Large Cliques in Sparse Graphs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 243–253.

[22] Aristides Gionis and Charalampos E Tsourakakis. 2015. Dense subgraph discovery: KDD 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2313–2314.

[23] Felipe Glaria, Cecilia Hernández, Susana Ladra, Gonzalo Navarro, and Lilian Salinas. 2021. Compact structure for sparse undirected graphs based on a clique graph partition. *Information Sciences* 544 (2021), 485–499.

[24] 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. 2017. Quickly finding a truss in a haystack. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091038

[25] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[26] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast Triangle Counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms* (New Orleans, Louisiana) *(IA<sup>3</sup> '14)*. IEEE Press, 1–8.

[27] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. 2012. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 8 (2012), 1651–1660.

[28] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[29] Mahantesh Halappanavar and Sayan Ghosh. 2020. *TriC: Distributed-memory Triangle Counting by Exploiting the Graph Structure.* Technical Report. Pacific Northwest National Lab.(PNNL), Richland, WA (United States).

[30] CJ Henry and P Eng. 2014. A Parallel GPU Solution to the Maximal Clique Enumeration Problem for CBIR. In *GPU Technology Conference*.

[31] Yang Hu, Hang Liu, and H Howie Huang. 2018. High-performance triangle counting on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–5.

[32] Wen-Mei W Hwu, David B Kirk, and Izzat El Hajj. 2022. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann.

[33] Shweta Jain and C Seshadhri. 2020. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 268–276.

[34] John Jenkins, Isha Arkatkar, John D Owens, Alok Choudhary, and Nagiza F Samatova. 2011. Lessons learned from exploring the backtracking paradigm on the GPU. In *European Conference on Parallel Processing*. Springer, 425–437.

[35] Bernhard Kerbl, Michael Kenzel, Joerg H Mueller, Dieter Schmalstieg, and Markus Steinberger. 2018. 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*. 76–85.

[36] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

[37] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*. Springer, 303–336.

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

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

[40] Rong-Hua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering heuristics for k-clique listing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2536–2548.

[41] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2016. Network motif discovery: A GPU approach. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2016), 513–528.

[42] Amogh Lonkar and Scott Beamer. 2021. Accelerating Clique Counting in Sparse Real-World Graphs via Communication-Reducing Optimizations. *arXiv preprint arXiv:2112.10913* (2021).

[43] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan. 2018. Linear Algebraic Formulation of Edge-centric K-truss Algorithms with Adjacency Matrices. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547718

[44] Vikram S Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2018. Collaborative (CPU+GPU) algorithms for triangle counting and truss decomposition. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[45] Samuel Manoharan. 2020. Patient Diet Recommendation System Using K Clique and Deep learning Classifiers. *Journal of Artificial Intelligence* 2, 02 (2020), 121–130.

[46] Rasmus Pagh and Charalampos E Tsourakakis. 2012. Colorful triangle counting and a mapreduce implementation. *Inform. Process. Lett.* 112, 7 (2012), 277–281.

[47] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[48] R. Pearce and G. Sanders. 2018. K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547572

[49] Carl Pearson, Mohammad Almasri, Omer Anjum, Vikram S Mailthody, Zaid Qureshi, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on triangle counting on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[50] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*. 1431–1440.

[51] Ryan A Rossi, Nesreen K Ahmed, and Eunyee Koh. 2018. Higher-order network representation learning. In *Companion Proceedings of the The Web Conference 2018*. 3–4.

[52] Ryan A Rossi and Rong Zhou. 2018. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data* 5, 1 (2018), 1–14.

[53] Ryan A Rossi and Rong Zhou. 2019. System and method for compressing graphs via cliques. US Patent 10,217,241.

[54] Ryan A Rossi, Rong Zhou, and Nesreen K Ahmed. 2018. Estimation of graphlet counts in massive networks. *IEEE transactions on neural networks and learning systems* 30, 1 (2018), 44–57.

[55] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. 2009. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel and Distrib. Comput.* 69, 4 (2009), 417–428.

[56] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel clique counting and peeling algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 135–146.

[57] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

[58] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*. 1122–1132.

[59] Matthew VanCompernolle, Lee Barford, and Frederick Harris. 2016. Maximum Clique Solver using Bitsets on GPUs. In *Information Technology: New Generations*. Springer, 327–337.

[60] Phonexay Vilakone, Doo-Soon Park, Khamphaphone Xinchang, and Fei Hao. 2018. An efficient movie recommendation algorithm based on improved k-clique. *Human-centric Computing and Information Sciences* 8, 1 (2018), 1–15.

[61] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693* (2012).

[62] Leyuan Wang and John D Owens. 2019. Fast BFS-based triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[63] Leyuan Wang and John D Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. *arXiv preprint arXiv:2003.01527* (2020).

[64] Leyuan Wang, Yangzihao Wang, and John D Owens. 2016. Fast parallel subgraph matching on the GPU. In *HPDC*.

[65] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. 2016. A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing*. 1–8.

[66] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2017. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering* 30, 1 (2017), 73–86.

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

[68] Yanyan Xu, James Cheng, Ada Wai-Chee Fu, and Yingyi Bu. 2014. Distributed maximal clique computation. In *2014 IEEE International Congress on Big Data*. IEEE, 160–167.

[69] Peter Yamout, Karim Barada, Adnan Jaljuli, Amer E Mouawad, and Izzat El Hajj. 2022. Parallel Vertex Cover Algorithms on GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[70] Ting Yu and Mengchi Liu. 2019. A memory efficient maximal clique enumeration method for sparse graphs with a parallel implementation. *Parallel Comput.* 87 (2019), 46–59. https://doi.org/10.1016/j.parco.2019.05.005

[71] Yanlei Yu, Zhiwu Lu, Jiajun Liu, Guoping Zhao, and Ji-rong Wen. 2019. RUM: Network representation learning using motifs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1382–1393.

[72] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.

[73] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1249–1260. https://doi.org/10.1109/ICDE48307.2020.00112