Maximum Clique Enumeration on the GPU

Afton Geil
Dept. of Electrical & Computer Engr.
University of California, Davis
Davis, California, USA
angeil@ucdavis.edu

Serban D. Porumbescu
Dept. of Electrical & Computer Engr.
University of California, Davis
Davis, California, USA
sdporumbescu@ucdavis.edu

John D. Owens
Dept. of Electrical & Computer Engr.
University of California, Davis
Davis, California, USA
jowens@ece.ucdavis.edu

***Abstract*—We present an iterative breadth-first approach to maximum clique enumeration on the GPU. The memory required to store all of the intermediate clique candidates poses a significant challenge. To mitigate this issue, we employ a variety of strategies to prune away non-maximum candidates and present a thorough examination of the performance and memory benefits of each of these options. We also explore a windowing strategy as a middle-ground between breadth-first and depth-first approaches, and investigate the resulting tradeoff between parallel efficiency and memory usage. Our results demonstrate that when we are able to manage the memory requirements, our approach achieves high throughput for large graphs indicating this approach is a good choice for GPU performance. We demonstrate an average speedup of 1.9x over previous parallel work, and obtain our best performance on graphs with low average degree.**

***Index Terms*—parallel, GPU, maximum clique, graph algorithms**

I. Introduction

The maximum clique(s) of a graph is the largest group(s) of fully connected vertices. As one of Karp's 21 NP-complete problems, the maximum clique problem is among the most studied combinatorial problems in graph theory. While this problem has been widely studied from a theoretical point of view, it can also be a useful tool for many real world graph applications including: social network analysis [1], Internet topology analysis [2], and systems biology [3].

The most common approach to finding maximum cliques is a depth-first branch and bound algorithm, in which a new vertex is added to the clique-in-progress at each level of the search tree, and bounds on the best possible solution for each branch are computed at every branch point and compared to the current best clique found so far to determine which vertex to add next. Backtracking algorithms like these are notoriously difficult to implement efficiently on GPUs. When Jenkins et al. implemented the closely-related maximal clique enumeration problem on the GPU, they found that they could not achieve more than a modest speedup over a single-threaded CPU implementation due to challenges with high divergence, workload imbalance, and irregular memory access patterns [4].

In this paper, we focus primarily on maximum clique enumeration — finding *every* clique of the maximum size in a graph. Although most previous work has focused on finding just one of the maximum cliques, we believe that solving for all maximum cliques is more broadly useful. We highlight the following contributions:

- A breadth-first search approach to *maximum clique enumeration* on the GPU.
- A variety of methods to reduce memory via pruning, including different heuristics and traversal orderings.
- A data structure designed specifically for efficiently expanding many lists of vertices in parallel to track candidate cliques.
- A *windowed search* scheme (for finding a single, maximum clique when memory constraints prevent enumeration) which allows us to explore a middle-ground between a depth-first and breadth-first search and the tradeoffs between memory usage and available parallel work.
- A parallel heuristic which manages to find a clique of maximum size for 97% of the datasets in our test set before we even begin running the exact algorithm.

Our code will be open-sourced to enable future work on maximum clique and other graph algorithms on the GPU.

II. Background

Given a graph $G = (V, E)$, a *clique*, $C \subseteq V$, is a subset of vertices such that each vertex in C is connected to every other vertex in C via an edge, i.e., the subgraph induced by C is a *complete subgraph* of G. The maximum clique(s), C_ω, are the clique(s) with the largest cardinality. The size of the maximum clique is also known as the *clique number* of a graph, denoted as $\omega(G)$. Different applications may have use for the clique number on its own, the clique number and multiplicity, the list of the vertices belonging to one of the maximum cliques, or the members of all cliques of size ω.

A. The Search Tree

Since our aim is to find the *exact* maximum clique(s) of a given graph, we must use a systematic approach to consider all possible combinations of vertices in order to guarantee that we have found the largest set(s) of fully connected vertices. This problem is often solved using branch and bound algorithms, with the goal of swiftly eliminating most of these combinations via discerning choices of bounds and traversal order of the search tree. The basic branching algorithm is as follows: begin with an empty clique set, C, and a set of candidate vertices, P, which initially includes all vertices in G. Then, following some ordering scheme, select a vertex $v \in P$ to add to C, and filter out vertices in P not connected to v. Next, select another vertex remaining in P, filter again, and repeat until

P is empty, then note this clique and its size. Backtrack to the previous decision point, select a different vertex from the candidate set, and continue on, maintaining a record of the largest clique found so far, until all combinations have been exhausted. In the complete branch and bound algorithm, this search tree traversal is pruned by applying bounds at each branch point to reduce the number of unfruitful branches that are explored before returning the solution.

B. Bounding the Search

Most implementations use three bounds in pruning the search space: (1) a lower bound on the maximum clique size, (2) an upper bound on the largest clique a vertex belongs to, (3) an upper bound on the largest clique within each set of vertices.

1) Setting an Initial Lower Bound: The size of the largest clique found so far serves as the lower bound on the maximum clique size; however, a heuristic can be used to find a lower bound before beginning the search, in order to preprune the candidate list. Due to the computational complexity of the maximum clique problem, there is a substantial body of previous work on a wide variety of heuristics, which aim to avoid paying the cost of computing an exact solution. Selecting a heuristic involves a trade-off of work between preprocessing and the exact computation.

2) Pruning Individual Vertices: If we have an upper bound on the largest possible clique a vertex can belong to, then we can compare this against the largest clique found so far and determine whether or not the vertex could be a member of a larger clique. If not, we can ignore this vertex entirely. A simple upper bound for a vertex is its degree plus one. However, we can obtain a tighter bound using the concept of k-cores. A *k-core* of a graph is a vertex-induced subgraph in which all vertices have degree at least k [5]. The largest value of k for which a vertex is a member of a k-core is its *core number*. The largest clique a vertex could be a member of is its core number plus one. We compare the effectiveness of pruning using vertex degrees and core numbers.

3) Finding Upper Bounds for Sets of Vertices: As we traverse the search tree, we use an upper bound on the largest clique contained within the candidate set, P, in order to determine whether to continue to explore the branch or prune it. The most straightforward upper bound is $|C| + |P|$, the size of the current clique set plus the size of the candidate set. Alternatively, we can find a tighter upper bound using other metrics, such as vertex coloring.

C. GPU-Specific Considerations

When designing algorithms for GPUs, we must tailor our implementations to their unique architecture in order to achieve high performance. GPUs are optimized for high throughput, while CPUs are optimized for low latency. Because we have thousands of threads available for computation on a single GPU, we care less about work efficiency and more about maximizing available parallelism and how to best split this work up between threads. Ideally, work is distributed in a balanced way to take full advantage of the compute available. We should also avoid divergence between threads' execution paths, particularly threads within the same 32-thread grouping, known as a *warp* in the CUDA programming model. Threads in the same warp run in lockstep, so when some threads take a different execution path, the others are idling.

As described in Section II-A, the most common method for traversing the search tree is a depth-first approach with backtracking; however, these types of algorithms map poorly onto the massive parallelism of GPUs, due to a lack of available parallel work, high divergence, and imbalanced workloads [4]. If we choose a depth-first algorithm, we could traverse the search tree in a fine-grained thread-parallel or coarse-grained warp-parallel fashion. Both options present challenges for an efficient GPU implementation. In a fine-grained thread-parallel traversal, each thread is assigned its own subtree to search independently. Because the depth of subtrees is irregular and unpredictable, this leads to high divergence and an unbalanced workload. For a coarse-grained warp-parallel traversal, threads in each warp traverse the search tree as a group and work cooperatively to compute the new candidates and bounds at each branch point. Although this avoids the high divergence of the fine-grained traversal, it reduces the amount of parallel work available and does not provide enough work for all threads when the candidate list is shorter than warp-sized.

Another GPU optimization to keep in mind is that in order to maximize memory bandwidth, we should use coalesced memory accesses whenever possible – that is, we want neighboring threads to access values stored in a contiguous chunk of memory. Again, due to the irregular nature of the search tree, the length of candidate lists is highly variable, making it difficult to arrange coalesced memory accesses. Finally, GPU RAM size is limited, and in order to avoid the additional communication costs associated with out-of-core implementations, we aim to keep overall data use small enough to fit into GPU memory.

D. Breadth-First Strategy

As the basis of our implementation we chose a breadth-first exploration of the search tree in order to maximize the available parallelism, minimize divergence, and improve load balancing. In a breadth-first traversal, we take all branches at each level before moving deeper into the tree. When performing the search sequentially, this is not ideal, because the maximum cliques are found at the deepest leaves of the tree; however, the massively parallel nature of GPUs allows us to explore many of these branches simultaneously instead. Though it will likely require more work overall because we are not updating the lower bound throughout the computation, we can utilize the many available threads, so we hope this allows us to finish the entire search more quickly.

Although a breadth-first approach maximizes the available parallelism, the space required to store all cliques and candidates at once is a limitation of this approach. For a depth-first search, when we reach the end of a path, if the solution found is not a new maximum, the clique and its associated data are discarded. In a parallel breadth-first search, all branches are taken at once, so we need to store all k-cliques at each level of the tree, which may be impractical, particularly for large or dense graphs. In our work, we investigate ways to overcome these memory constraints via pruning and some deviations from the typical breadth-first traversal.

III. Related Work

Maximal cliques are cliques not contained within a larger clique. The *maximum clique(s)* of a graph is the largest of the maximal cliques. The search trees for finding maximal or maximum cliques are similar, but because maximal cliques can be of any size, it is not possible to use bounds to prune the search. However, similar techniques for parallelizing the search may be useful in both problems. Previous parallelizations of maximum clique and maximal clique enumeration algorithms have primarily targeted multi-threaded or distributed CPU systems, though there have also been a few GPU implementations.

1) Search Tree Traversal: Most parallelizations of maximal and maximum clique algorithms have taken a depth-first approach, including implementations on both CPUs [1], [2], [6], [7] and GPUs [4], [8]. Though we have found no previous breadth-first maxim*um* clique implementations, a couple implementations of maxim*al* clique enumeration on CPUs [3] and GPUs [9] use a breadth-first traversal to increase available parallelism and easily output maximal cliques in order of increasing size. For these implementations, the authors find that the memory requirements for a breadth-first approach limit the size and/or density of the graphs they are able to solve. In their maximal clique enumeration implementation on the GPU, Wei et al. take an interesting middle ground approach, exploring one broader subtree at a time in parallel [10]. To avoid running out of memory, they select the width of this subtree based on an upper bound on the number of maximal cliques from Moon and Moser's theorem [11]. In our work, we use a similar technique of exploring a smaller subtree (windowing), and because we are able to prune many of the candidate cliques, the memory requirements for storing the candidates in each subtree are smaller.

2) Parallelization: Previous CPU maximum clique implementations have used a fine-grained thread-parallel traversal, which does not translate well to GPUs, as described in Section II-C. GPU maximal and maximum clique implementations using a coarse-grained warp-parallel traversal have suffered from insufficient parallel work and imbalanced workload [4], [8]. Breadth-first implementations can instead utilize an iterative data-parallel approach [9], which tends to be better suited to massive parallelism available on GPUs. This is the approach we chose for both the exact computation and all preprocessing, because it allows us to match the parallelism to the problem size at each stage. Once again, Wei et al. chose a mixed approach, using data-parallel operations for some parts of the computation and warp-parallel operations for others [10].

3) Data Structures: In order to efficiently utilize memory, we must make careful choices in the data structures we use for storing the graph and candidate cliques. Some CPU implementations create one copy of the graph per worker [1], [6], or partition the graph and give one partition to each worker [2]. Due to the limited memory size per thread, these are not practical options for a GPU implementation. The choice of graph data structure determines the speed of computing set intersections for filtering out unconnected vertices. The fastest intersections use bitwise operations [3], [7], [8]; however, this requires storing the graph as an adjacency matrix, which is very space-inefficient. An adjacency list structure, such as compressed sparse row format (CSR), saves memory, but requires either linear or binary (if sorted) searches for each edge. A middle ground option is to use hash tables to compute set intersections [9]. In some previous work different data structures are chosen based on the size of the input graph and amount of available memory [1], [4], [6]. Because our work is aimed at larger graphs and we have limited memory space on the GPU, we chose a CSR with sorted adjacency lists and binary searches for set intersections.

For breadth-first implementations, we also need a data structure for storing the large number of candidate cliques. Lessley et al. use hash tables for storing cliques and computing set intersections [9]. Wei et al. store the subgraphs for each active subtree using a CSR-like representation with labels for which set (current clique, candidate, already-explored) each vertex belongs to [10]. In this work, we introduce a new data structure for storing intermediate candidate cliques, which is compact and allows for coalesced memory accesses.

4) Bounds and Pruning: Previous parallel maximum clique implementations use coloring [2], [7], [8] or core numbers [1] to prune the search tree and choose the order of traversal. We compare the effect of using core numbers and vertex degrees for pruning and ordering the search. Rossi et al. also use a parallelized greedy heuristic to set an initial lower bound and improve pruning in earlier stages of the search [1]. We also use this approach, which is particularly important for a breadth-first implementation, because the lower bound is not improved throughout the search, as it is in a depth-first implementation. We hope that by pruning the search using a high-quality initial lower bound, we are able to sufficiently mitigate the memory requirements of a breadth-first traversal.

IV. Implementation

We find the maximum cliques by performing a breadth-first traversal of the search tree via an iterative process. In each iteration, we launch one thread per candidate vertex across all of the candidate lists in the current level. Each thread adds its vertex to its the clique set and generates the list of candidates for the next level of the search. We wait until all threads have finished, then repeat the process for the next level of the search tree.

The steps of our implementation are as follows: (1) (optionally) compute the vertex k-core decomposition of the graph, (2) find an initial lower bound maximum clique via a greedy heuristic, (3) form the initial lists of 2-cliques/candidates, (4) perform the iterative process described above, adding vertices to the clique lists and generating new candidate lists for the next iteration. Each of these steps is performed in parallel on the GPU. We use the graph loader from the Gunrock GPU graph library [12] in preprocessing to convert the input dataset into CSR format, which we store in GPU global memory to utilize throughout the rest of the computation. For the vertex k-core decomposition, we use the implementation from the Gunrock applications examples. We also make use of NVIDIA's CUB library throughout our implementation for its optimized scan, reduce, select, and sort operations [13]. In this section, we describe the details of our implementation, as well as a modified version of the breadth-first approach, in which we explore only a subset of the candidates at a time, which we refer to as a windowed breadth-first search.

A. Heuristic

A heuristic is used to establish a lower bound on the maximum clique size. As described in Section II-B1, the choice of heuristic involves a tradeoff of work between preprocessing and the exact algorithm. We selected a greedy heuristic rather than a more complicated heuristic because we are aiming to minimize preprocessing time, and we expect that the GPU can handle a large amount of work in the exact algorithm stage. We have two different implementations of the greedy heuristic: (1) the *single run version* in which we run the greedy algorithm once and use the GPU threads to filter the vertex list in parallel and (2) the *multi-run version* where we run many instances of the greedy algorithm in parallel on the GPU. For both versions, we provide an option to use either the vertex degrees or core numbers for determining the greedy ordering.

1) Single Run Heuristic: The greedy heuristic is as follows: start with a list of all vertices and pick the vertex with the highest degree (or core number) to add to the clique-in-progress, then remove any vertices not connected to this vertex. From the remaining vertices, add the vertex with the highest degree (or core number) to the clique and once again filter out any vertices not connected to this vertex. Repeat until no vertices remain in the list. The size of the clique found this way serves as a lower bound on the maximum clique size.

In our GPU implementation of this heuristic, we first create a list of all vertices in the graph and use the GPU to sort the vertices descending degree (or core number) order. We pull the first vertex, v_0, out of the candidate list and filter the vertex list on the GPU using a parallel select operation, removing any vertices which are not neighbors of v_0. Then we pick the next vertex from the filtered candidate list, and filter the list again. This process repeats until there are no vertices remaining in the list. The number of iterations of this greedy algorithm is the lower bound clique size, $\overline{\omega}$.

2) Multi-Run Heuristic: For the multi-run greedy heuristic, we use the same greedy algorithm as the single-run heuristic, except that we run many instances of it in parallel, each with a different starting vertex. The implementation makes use of a variety of data-parallel operations which are well-suited to the GPU. The details are shown in Algorithm 1. As in the single run version, we begin with a list of all vertices sorted by decreasing degree/core number, and also a list of the vertices' degrees/core numbers, which we use to select the next vertex to add to each of the cliques-in-progress in each iteration. We select the number instances of the heuristic we would like to run, $h \leq |V|$, and use the h vertices with the highest degree/core numbers as the seed vertices for each of the runs. We create segmented arrays containing all of the neighbor vertices and their degrees/core numbers for each of the seeds. Then we begin to iterate. First, we find the vertex in each segment with the highest degree/core number using a segmented max operation. We use one thread per segment to flag each vertex in the segment that is connected to this vertex. Next, we use a select operation to filter the vertex and degree/core number arrays, removing vertices that are not connected. Then we remove empty segments with one more select operation and update the segment indices via a scan operation. We iterate until there are no candidate vertices remaining in any of the segments. As in the single run version, the number of iterations is the lower bound on the maximum clique size, but in this case it represents the largest clique found across all h parallel runs of the greedy heuristic. We expect that using the best of multiple runs will result in a better lower bound and, therefore, better pruning.

B. Clique List Data Structure

An important consideration for our breadth-first parallel implementation is how to store all of the cliques and candidate lists. In parallel in each iteration, we are creating a new candidate list for each of the current candidate vertices across all candidate lists. The size of each of these new candidate lists can vary widely between cliques and between iterations of the algorithm, making it impossible to preallocate the appropriate amount of memory. As mentioned in Section II-D, memory size is a significant concern for the breadth-first implementation, so we would also like to store cliques and candidate vertices as compactly as possible and avoid storing any duplicate information.

a) Criteria: The goal is to build a minimally-sized data structure that supports the following operations: (1) add a variable number of total items in each iteration, (2) track which clique each of the newly-added candidate vertices belongs to, and (3) delete data for cliques that have been pruned. Parallel operations take place with one thread per candidate vertex, so we would like to store all candidates in a contiguous block of memory to achieve coalesced accesses.

b) Our Solution: The data structure we chose, which we call a *clique list*, is essentially a linked list wherein each node of the list contains a pair of arrays, `vertexID` and `sublistID`. Figure 1 shows the clique list for an example graph. Each node in the clique list contains all of the necessary data for one iteration of the search. `vertexID` contains the candidate vertices for that level, and `sublistID` contains the index in the previous clique list node where the last vertex added to the clique is stored. Effectively, the `sublistID` is a pointer into the previous clique list node's `vertexID` array. The `sublistID` array allows us to identify which

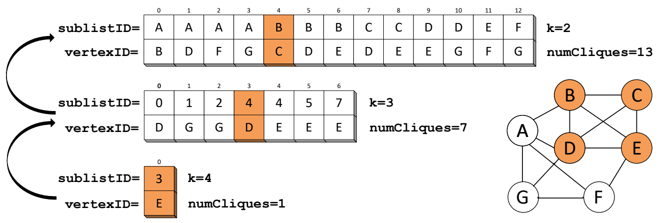


Fig. 1: An example graph and the clique list structure used to find its maximum clique. To illustrate how the clique list structure works, let's walk through how to read out the maximum clique, represented by the circled blocks. We start at the head of the list, which is the most-recently added node, for $k = 4$. There is only one clique of size 4, so this graph only has one maximum clique. $\texttt{vertexID}_4[0] = E$ means vertex E is in the clique, and $\texttt{sublistID}_4[0] = 3$, so we follow the $\texttt{previous}$ pointer to the clique list node for $k = 3$ and find $\texttt{vertexID}_3[3] = D$. Now we have the clique set $C = \{E, D\}$, and we use the $\texttt{sublistID}_3[3] = 4$ as a pointer into the clique list node for $k = 2$, where both $\texttt{vertexID}_2[4] = C$ and $\texttt{sublistID}_2[4] = B$ represent vertices in the clique. Therefore, the maximum clique of this graph is $C = \{E, D, B, C\}$.

vertices belong to the same candidate list and, at the end of the computation, to read out all of the vertices in the maximum clique(s). The first node of the clique list is different from the others. Because there is no need for indices into a previous node's vertex array, we combine the data for the first two levels of the search tree into one node by using `sublistID` to store the vertex IDs for the first level of the tree.

c) Discussion: This data structure allows us to simultaneously expand all cliques in each iteration, allocating memory as needed, and to track which vertices belong to each clique. Iterating through the linked list to read out the clique vertex sets is cumbersome, but within each iteration of the search, we only need to know the current candidate vertices in order to check their connections and generate the candidate list for the next iteration; therefore, we only need to access values in previous nodes of the linked list at the end of the computation to read out the members of the maximum clique(s). We avoid storing duplicate information because each of the k-cliques generated from the same $(k-1)$-clique points back to their shared ancestor in the previous clique list node. This structure also allows for coalesced accesses, because neighboring threads read from consecutive values in the `vertexID` and `sublistID` arrays. One drawback of this data structure is that it is very difficult to delete data for cliques that have been ruled out as candidates for the maximum clique, because `sublistID` values would need to be updated in all nodes of the clique list structure. We could not find another data structure that met our other criteria and allows for simpler deletions, so we accept this downside and do not delete any data for eliminated cliques.

C. Setup: Forming the 2-Clique List

Before running the breadth-first search, we must set up the first node of the clique list containing all of the 2-cliques. This is essentially a list of the edges in the graph stored as an array of source vertices and an array of destination vertices. We create the 2-clique list on the GPU using data-parallel operations. The steps are: (1) one thread per vertex determines the number of neighbors in its sublist (2) prune sublists shorter than lower bound clique size determined in heuristic (3) a scan operation to determine start indices for sublists and amount of memory to allocate for 2-clique list (4) one thread per unpruned sublist outputs the vertices for that sublist.

In order to avoid storing duplicate cliques and wasting work throughout the rest of the computation, we use only one of the two directed edges that represent each undirected edge in the original graph, known as an *orientation* of the graph. We do not explicitly modify the graph, but instead select the desired edges when forming this initial clique list node. From each reciprocal edge pair, we keep the edge where the source vertex has lower degree. Orienting the graph by degree improves pruning over orientation by index, because vertices with lower degree have shorter adjacency lists. Selecting these vertices as the source vertices means that the initial sublists are shorter on average and thus a greater proportion of sublists will be smaller than $\bar{\omega}$. In addition to pruning entire sublists, we also pre-prune individual vertices by comparing their degree/core number to the lower bound found in the heuristic. After performing all pruning, it is possible that there may be only one sublist remaining, representing the clique discovered by the heuristic. If this is the case, we can skip the full exact algorithm because we have already found the singular maximum clique.

The final preprocessing operation is to sort vertices by degree within their candidate lists. Without this sort, vertices will be in the same order as they are stored in the adjacency lists — sorted in order of increasing index values. By sorting vertices according their degrees, we hope to improve pruning, because the vertices near the beginning of the candidate lists are assigned more edge lookups than vertices later in the list. This means lookups for missing edges are moved to earlier iterations, enabling us to prune them earlier in the search. Additionally, placing the low degree vertices at the beginning of the candidate lists means a greater fraction of edge lookups are in shorter adjacency lists, which reduces the average lookup speed.

D. Breadth-First Maximum Clique

The iterative breadth-first search, detailed in Algorithm 2, now proceeds as follows: one thread for each vertex in the clique list checks whether it is connected to each of the vertices that follow it in its sublist, and tallies the number of successful edge lookups. Each successful lookup represents a $(k+1)$-clique. The length of this new sublist is compared to $\bar{\omega}$ to determine whether it should be pruned before returning the count. Next, we use a scan operation to find the start indices for the new sublists and amount of memory to allocate for the $(k+1)$-clique list. If there are no new cliques, we have reached the end of the search, and we use the current clique list node to read out the vertices in the maximum clique(s). Otherwise, we assign one thread per vertex to output the candidate vertices for its new sublist.

This breadth-first approach provides the ability to easily launch a different number of threads in each iteration to match the number of candidate vertices formed in the previous iteration. This avoids the load imbalance of having each thread traverse multiple levels of the search tree, which would result in both many dead-end threads and threads with vast search trees to explore. The largest portion of the computation in each iteration is the edge checks, each of which consists of a binary search (lines 5 and 19) on the candidate vertex's adjacency list within the CSR. Unfortunately, these memory accesses will not be coalesced, because neighboring threads are responsible for different candidate vertices. However, individual threads may receive some benefit from caching, because all of their reads will be in the same part of the graph data structure.

E. Windowed Search

As mentioned in Section II-D, one of the challenges for a breadth-first implementation is the large memory requirement for storing all candidates simultaneously. Particularly for large and/or dense graph datasets, there may be more candidate cliques than can fit in GPU memory, even after pruning. For these instances, we consider an approach for solving for only one maximum clique, rather than enumerating all maximum cliques. We implement a windowed variation on the breadth-first search, wherein we split up the initial list of 2-cliques and run our breadth-first maximum clique algorithm on one subset (window) of candidates at a time. Although a fully depth-first search provides little parallelism and creates too much divergence and workload imbalance between threads to perform well on a GPU, we hope that modifying the search to be less broad can offer a balance between parallelism and memory requirements.

a) Implementation: We want to ensure that the window boundary is between sublists, since candidate vertices use the information for all vertices that follow them in their sublist. To find the end of a sublist closest to the nominal end of the window, we use the GPU threads to quickly read a chunk of `sublistID` values and check if their index is the end of a sublist, and if so, write their index to a global variable using an `atomicMin`. When we have finished searching one window, we update the lower bound if a new largest clique has been found, find the tail for the next window, and repeat until we have finished all windows. With windowing, we have the ability to choose an ordering for the search, as other depth-first implementations do. We experiment with sorting the source vertices in the 2-clique list by their degrees or core numbers and describe our findings in Section V-C.

b) Discussion: It is still possible that the combinations from a relatively small set of 2-cliques can lead to a very large list of candidates for larger cliques. The choice of window size is important, because we want to provide enough work to keep the GPU busy, but keep the clique list small enough to stay within memory bounds. We expect graphs with higher average degree to work best with a smaller window, because the number of candidates will probably increase more quickly with each iteration. We test a variety of window sizes and traversal orderings and describe the trade-offs we find in Section V-C.

Algorithm 1 Multi-Run Greedy Heuristic

```
1: function GETNEIGHBORCOUNTS($G$, $vertices$, $neighborCounts$)
2:     $neighborCounts[\texttt{threadID}] \leftarrow |N(vertices[\texttt{threadID}])|$

3: function SETUPNEIGHBORTHRESHOLDS($G$, $vertices$, $neighborCounts$, $vertexThresholds$, $indices$, $neighbors$, $neighborThresholds$)
4:     $offset \leftarrow indices[\texttt{threadID}]$
5:     $count \leftarrow 0$
6:     for $u \in N(v)$ do
7:         $neighbors[offset + count] \leftarrow u$
8:         $neighborThresholds[offset + count] \leftarrow vertexThresholds[u]$
9:         INCR($count$)

10: function CHECKCONNECTIONS($G$, $neighbors$, $indices$, $maxIndices$, $flags$, $connectedCounts$)
11:     $v \leftarrow neighbors[maxIndices[\texttt{threadID}]]$
12:     $currentIndex \leftarrow indices[\texttt{threadID}]$
13:     $segmentEnd \leftarrow indices[\texttt{threadID} + 1]$
14:     $count \leftarrow 0$
15:     while $currentIndex < segmentEnd$ do
16:         $u \leftarrow neighbors[currentIndex]$
17:         if $u \in N(v)$ then
18:             $flags[currentIndex] \leftarrow$ TRUE
19:             INCR($count$)
20:         else
21:             $flags[currentIndex] \leftarrow$ FALSE
22:         INCR($currentIndex$)
23:     $connectedCounts[\texttt{threadID}] = count$

24: function MULTIRUNGREEDYHEURISTIC($G$, $vertices$, $vertexThresholds$, $h$)
25:     ▷ $vertices$ sorted in order of descending degree or core number
26:     for all $vertices$ do
27:         GETNEIGHBORCOUNTS($G$, $vertices$, $neighborCounts$)
28:     $indices$ ←CUBSCAN($neighborCounts$)
29:     for all $vertices$ do
30:         SETUPNEIGHBORTHRESHOLDS($G$, $vertices$, $neighborCounts$, $vertexThresholds$, $indices$, $neighbors$, $neighborThresholds$)
31:     $numSegments \leftarrow h$
32:     $\bar{\omega} \leftarrow 1$
33:     while $numSegments > 0$ do
34:         $maxIndices$ ←CUBSEGMENTEDMAX($neighborThresholds$)
35:         for all segments do
36:             CHECKCONNECTIONS($G$, $neighbors$, $indices$, $maxIndices$, $flags$, $connectedCounts$)
37:         ($neighbors$, $numCandidates$) ←CUBSELECT($neighbors$, $flags$)
38:         ($neighborThresholds$, $numCandidates$) ←CUBSELECT($neighborThresholds$, $flags$)
39:         if $numCandidates = 0$ then
40:             break
41:         ($nonzeroCounts$, $numSegments$) ←CUBSELECTIF($connectedCounts$)    ▷ keep values $> 0$
42:         $indices$ ←CUBSCAN($nonzeroCounts$)
43:         INCR($\bar{\omega}$)
44:     return $\bar{\omega}$
```

V. Results

We evaluate our maximum clique implementation on the 58 largest real-world datasets (all datasets with $|E| > 10\text{k}$)[1] evaluated in Rossi et al.'s paper [1], downloaded from the Network Repository [14]. These include social, web, road, biological, technological, and collaboration networks ranging in size from 10k to 106M edges. We use Rossi et al.'s Parallel Maximum Clique (PMC) [1], a multi-threaded CPU implementation, as our main comparison, because no code is publicly available for previous GPU maximum clique implementations. We also note that Rossi's implementation only finds one of the maximum cliques, while our implementation enumerates *all* cliques of maximum size. We randomize the vertex indices, to avoid any bias from the ordering of the original datasets that could affect the comparisons for sorting by index and degree. We also preprocess the datasets (before forming the CSR data structure) to ensure all graphs are undirected and contain no loops. We run all GPU and CPU experiments on a Linux

[1]Our implementation is OOM for two datasets (friendster and flickr), so they do not appear in performance data, but are included in Table I.

Algorithm 2 Breadth-First Maximum Clique Enumeration

```
function COUNTCLIQUES($G$, $cliqueList_k$, $\bar{\omega}$, counts)
    $i \leftarrow$ threadID $+ 1$
    connected $\leftarrow 0$
    while $sublistID_k$[threadID] $=$ $sublistID_k[i]$ do
        if $vertexID_k[i] \in N(vertexID_k$[threadID]$)$ then
            INCR(connected)
        INCR($i$)
    if connected+k$<\bar{\omega}$ then                        ▷ pruning by sublist length
        connected $\leftarrow 0$
    counts[threadID] $=$ connected
    return

function OUTPUTNEWCLIQUES($G$, $cliqueList_k$, offsets, $cliqueList_{k+1}$)
    $i \leftarrow$ threadID $+ 1$
    cliqueOffset $\leftarrow$ offsets[threadID]
    if cliqueOffset $=$ offsets[threadID $+ 1$] then
        return
    count $\leftarrow 0$
    while $sublistID_k$[threadID] $=$ $sublistID_k[i]$ do
        if $vertexID_k[i] \in N(vertexID_k$[threadID]$)$ then
            $vertexID_{k+1}$[cliqueOffset $+$ count] $\leftarrow$ $vertexID_k[i]$
            $sublistID_{k+1}$[cliqueOffset $+$ count] $\leftarrow$ threadID
            INCR(count)
        INCR($i$)
    return

function MAXCLIQUES($G$, $\bar{\omega}$, $cliqueList_k$, $cliqueCount_k$)
    $k \leftarrow 2$
    while $cliqueCount_k > 1$ do
        for all candidates in $cliqueList_k$ do
            COUNTCLIQUES($G$, $cliqueList_k$, counts)
        offsets $\leftarrow$CUBSCAN(counts)
        $cliqueCount_{k+1} \leftarrow$ offsets[$cliqueCount_k - 1$]
        if $cliqueCount_{k+1} = 0$ then
            break
        for all candidates in $cliqueList_k$ do
            OUTPUTNEWCLIQUES($G$, $cliqueList_k$, offsets, $cliqueList_{k+1}$)
        if $cliqueCount_{k+1} = \bar{\omega} - k + 1$ then
            break                              ▷ maximum clique was found by heuristic
        INCR($k$)
    return $cliqueList_k$
```

workstation with a 2.8GHz 24-Core AMD EPYC 7402 CPU and 512GB of main memory and an NVIDIA Tesla A100 GPU with 40GB of on-board memory. Our code[2] is compiled with CUDA 11.6. For both the overall throughput results and comparison with PMC in Section V-A, we report the results from the fastest configuration (for our implementation: the best combination of heuristic, window size, and other preprocessing; for PMC: the best number of threads) for each dataset. Reported runtimes for our implementation and PMC represent the average of 5 runs, and do not include the time to load the graph dataset onto the GPU, but do include the heuristic runtime and other preprocessing.

Key Takeaways:

- Our implementation performs best for larger graphs with low average degree.
- We achieve significant speedups over PMC on low degree graphs, while they tend to be faster for high degree graphs.
- For some graphs with high average degree and other hard to prune graphs, our implementation runs out of memory for storing candidate cliques.
- Breaking the search up into smaller windows does enable us to solve more hard to prune graphs, but at a significant performance cost.
- Better pruning does not dependably improve runtimes, so the goal is to minimize preprocessing time and prune just well enough to avoid running out of memory.
- The overall best heuristic is the multi-run degree-based heuristic. Smaller graphs perform best with a simple heuristic, while the hardest to prune graphs perform best with the multi-run core number heuristic.

A. Overall Performance

a) Performance vs Average Degree: **The most consistent factor determining the performance of our implementation is the average degree of the graph.** As shown in Figure 2, the number of edges processed per second decreases as the average vertex degree increases. There are a few factors at play here. First, graphs with higher degree are harder to prune because many of the vertices' degrees (or core numbers) will be larger than the heuristic lower bound. Therefore, candidates stick around for more iterations of the exact algorithm, requiring more work. Second, vertices in high degree graphs have larger adjacency lists, which corresponds to longer sublists in our algorithm. The workload for a thread in each iteration is dependent on the length of its sublist and its position within the sublist. With longer sublists, each iteration of the main loop has a longer runtime, and there is greater divergence and poorer load balancing amongst threads. Third, because each edge lookup requires a binary search, larger adjacency lists increase the work for each of these operations.

b) Performance vs Graph Size: **We achieve higher throughput as the graph size increases; however, the challenge is to avoid running out of memory (OOM) while solving these larger graphs, while also maintaining enough work to keep the GPU busy throughout the entire computation.** Figure 3 demonstrates that the runtime per edge decreases as the number of edges increases. This indicates that the GPU is able to handle these additional edges, and we may be able benefit from still greater efficiencies when scaling up to larger (but still low degree) graphs. However, when solving these larger graphs, particularly if they do not also have a very low average degree, there may be too many intermediate candidate cliques to fit in GPU memory. The solution to this problem is to improve pruning, but over-pruning leaves the GPU under-utilized in the early and late iterations, when there are fewer candidate cliques. So the key to optimal GPU performance is keeping the peak low enough to stay in GPU memory, while still leaving enough work in the early and late iterations to fill the GPU.

c) Comparison with Previous Work: **We find that our implementation outperforms PMC for low degree graphs, while PMC is faster for high degree graphs**, as shown in Figure 4. In general, the performance of their implementation is more dependent on the number of edges in the graph than the average vertex degree, while ours has the opposite trend. PMC is uses a fine-grained parallel depth-first implementation, which makes efficient use of the powerful threads available on the CPU, but the smaller number of threads means tbe maximum throughput is less than what can be achieved on the GPU. Therefore, **our implementation is more perfor-**

[2]Our implementation is available at https://github.com/owensgroup/GPUMaximumClique.

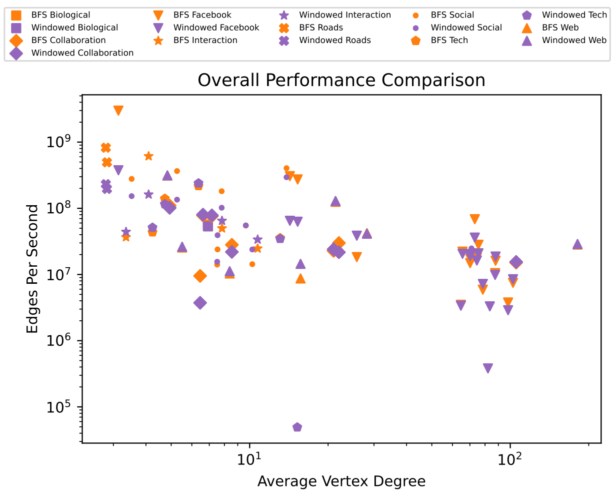


Fig. 2: Throughput (including all preprocessing) for fastest configuration on each dataset for basic breadth-first version and version with windowing. For both the regular breadth-first and windowed versions, performance is inversely correlated with average vertex degree.

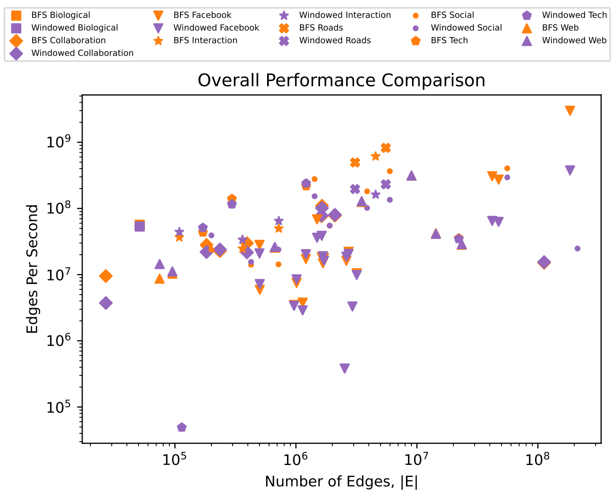


Fig. 3: Throughput (including all preprocessing) for fastest configuration on each dataset for basic breadth-first version and version with windowing. For both the regular breadth-first and windowed versions, throughput is higher for larger graphs.

mance scalable, except for the memory requirements of our breadth-first implementation. Adding the windowing option to our implementation does help to mitigate the memory requirements of the breadth-first implementation, but it comes at a performance cost. As you can see at the bottom of Figure 4, for the handful of graphs where only the windowed version is successful, PMC is significantly faster. Dividing the problem up into small enough windows to keep the memory requirements manageable tends to reduce the amount of parallel work so much that we cannot take advantage of the parallel power of the GPU.

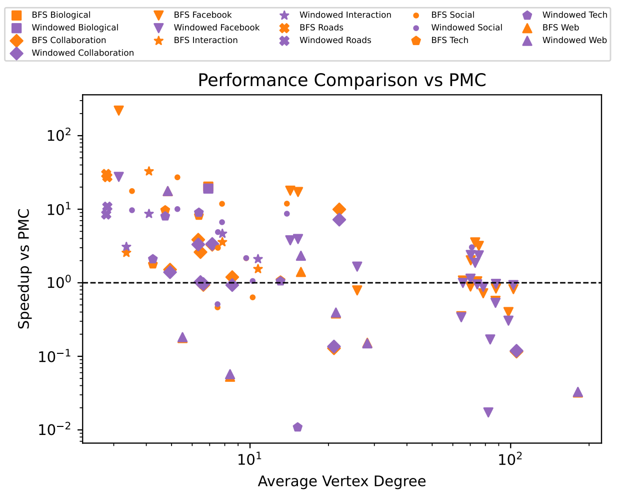


Fig. 4: Speedup over Rossi PMC for the fastest configurations of our regular breadth-first and windowed implementations.

B. Heuristics

For some small and/or low average degree graphs, it is possible to run the exact maximum clique computation without computing an initial lower bound. However, we find that without a heuristic lower bound to prune the search, we typically run out of memory when attempting to solve larger and/or higher degree datasets. As described in Section IV-A, we implemented four different versions of greedy heuristics. In these experiments, the multi-run heuristics use all vertices in the graph as seeds, i.e., $h = |V|$. We analyze the comparative effectiveness of the heuristics by comparing them along three metrics. (1) *Accuracy*: how close is the estimated lower bound to the true maximum clique size? (2) *Pruning*: how much memory do we save when pruning using these lower bounds, and is this sufficient to avoid OOM? (3) *Speedup*: does this pruning result in an overall speedup, or is the additional preprocessing time greater than the time saved in the exact computation?

1) Accuracy: **Of our four heuristic options (single-run degree, single-run core number, multi-run degree, and multi-run core number) the multi-run versions provide much better lower bounds than the single-run options.** Table I summarizes the mean error in the heuristic clique size across all datasets for each of our heuristics and the heuristic used in Rossi et al.'s PMC. Figure 5a shows how the runtime for each of our heuristics varies across graphs of different sizes. For the single-run versions, using core numbers does usually improve the the lower bound significantly, but it comes at the cost of a much longer runtime. The multi-run degree and multi-run core number heuristics result in similar accuracy, with the degree version finding a larger clique for some datasets and the core number version finding a larger clique for others, and for many datasets both versions succeed in finding a clique of the maximum size. Overall, the lower bounds from our multi-run heuristics are comparable to those of the heuristic used in PMC, which uses a similar algorithm

TABLE I: Comparison of heuristic error and number of graphs solvable (out of 58 total) using full breadth-first maximum clique

Heuristic	Mean Error	Solved Graphs	OOM
None	100%	19	67.2%
Single-run degree	63.3%	21	63.8%
Single-run core number	40.6%	35	39.7%
Multi-run degree	3.9%	50	16.0%
Multi-run core number	3.0%	47	10.3%
Rossi PMC	2.5%	58	0%

TABLE II: Geometric mean overall speedups comparison for different heuristics. Speedup numbers represent the performance improvement from using the heuristic listed in the column value over the baseline listed on the left of each row.

Baseline	Single Deg	Single Core	Multi Deg	Multi Core
None	$1.0x$	$0.4x$	$1.1x$	$0.4x$
Single Degree	—	$0.2x$	$0.3x$	$0.1x$
Single Core	—	—	$2.9x$	$1.1x$
Multi Degree	—	—	—	$0.9x$

to that of our multi-run core number heuristic.

2) Pruning Quality: **We find that the multi-run heuristics provide the best pruning and allow us to solve more datasets without running out of memory; however, graphs where the average degree is close to or larger than the maximum clique size are difficult to prune, even with an accurate lower bound.** Figure 5b shows that the quality of the lower bound is the main determining factor in achieving high levels of pruning. The multi-run heuristics achieve both the highest accuracy and the largest fraction of candidate cliques pruned. However, there are some datasets where even an accurate lower bound does not allow us to prune the candidates very aggressively. In our analysis, we found that in cases when the lower bound is not significantly larger than the average degree, pruning is less effective. This makes sense, because all of the upper bounds used in pruning are related to degree. Candidate lists are pruned based on their lengths, which are determined by the lengths of the vertices' adjacency lists. Vertices are pre-pruned based on their degree (or core number, which is typically correlated with degree). In instances where the heuristic finds one of the maximum cliques there is no way to increase pruning by improving the heuristic.

We do find that the improvement in pruning from more complex heuristics enables us to solve more datasets without going OOM, as shown in Table I, though these heuristics do typically have longer runtimes. If increased pruning also results in a faster runtime for the exact algorithm, this presents a tradeoff between preprocessing and main algorithm runtime. However, because our implementation runs in parallel for a fixed number of iterations, pruning the candidate lists past a certain point will not significantly improve the runtime of the exact algorithm, because we are not utilizing the full capacity of the GPU.

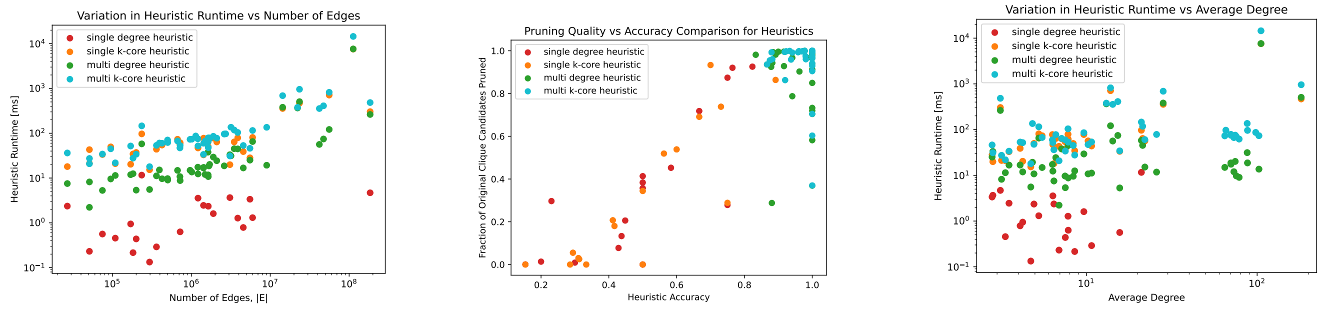
3) Speedup: **For graphs that can be solved without using a heuristic, it is typically fastest to to skip the heuristic altogether or use the single run degree heuristic. For larger or more dense graphs, the multi-run heuristics do provide speedups over the single run core number heuristic, and the multi-run degree heuristic is usually the fastest option.** Table II shows speedups for the breadth-first version when switching from less complex heuristics to the more complex ones (order of simplest to most complex: none, single-run degree, single-run core number, multi-run degree, multi-run core number). Because for many datasets, our implementation is OOM when we run it with no heuristic or one of the less accurate heuristics, we must use four different baselines in our comparison. The values listed in each row are the geometric mean speedups across all the datasets that require that baseline heuristic in order to complete the maximum clique computation without running out of memory. I.e., the datasets represented in the "None" row of Table II correspond to those counted in the "None" row of Table I.

a) No Heuristic Baseline: When no heuristic is *required* to run the maximum clique computation without running OOM, using the single run degree heuristic to set an initial lower bound can provide a small speedup. For this group of datasets, it is the ones with large maximum clique sizes that are more likely to get a speedup when adding a heuristic for the additional pruning. This is because the work saved in each iteration adds up over many iterations of the exact algorithm.

b) Single Run Core Number Heuristic Baseline: For graphs that can run with the single run core number heuristic, the multi-run degree heuristic delivers the best performance. Graphs with larger maximum clique sizes receive the biggest speedups with the multi-run degree heuristic over the single run core number heuristic. There are two reasons for this. (1) Each iteration of the multi-run heuristic requires more work than that of the single-run heuristic. (2) These datasets have maximum cliques that are considerably larger than the next largest clique, so as long as the heuristic succeeds in finding (one of) the maximum clique(s), the search is easy to prune. All three heuristics (single run core number, multi-run degree, multi-run core number) succeed at finding a clique of size ω for these graphs and achieve a high level of pruning.

c) Multi-Run Degree Heuristic Baseline: For graphs that require the multi-run heuristics to avoid OOM, about half run faster with the degree heuristic and half run faster with the core number heuristic. These are almost all Facebook datasets, which tend to have average degree higher than their maximum clique size, and are therefore hard to prune. This makes the high accuracy of the multi-run heuristics essential, and the additional accuracy and tighter vertex pruning upper bounds from the core numbers more likely to be beneficial.

4) Recommendations for Selecting a Heuristic: **As a general rule, the fastest runtime is typically achieved by using the simplest heuristic for which pruning is sufficient to avoid running out of memory. The best default choice for an unknown dataset is the multi-run degree heuristic.** For graphs with fewer edges and/or lower degree, likely no heuristic will be needed, while larger and higher degree graphs



(a) The runtime of each of our heuristics increases with the number of edges in the graph.

(b) The pruning quality for each of our heuristics is typically correlated with its accuracy.

(c) The runtimes for our heuristics are not correlated with the average degree of the graph.

Fig. 5: Overall performance results for our breadth-first and windowed maximum clique implementations.

will benefit from the multi-run heuristics. Figures 5a and 5c show that heuristic runtime increases with the number of edges, but not with the average degree. This further supports the conclusion that a more complex heuristic is likely to be beneficial for graphs with high degree.

As can be seen in Figure 5a, the k-core vertex decomposition increases the heuristic runtime significantly. Choosing a different k-core implementation could reduce the cost of this operation. However, since the core numbers only increase the accuracy by an average of 0.9% for the multi-run implementation, even a fast k-core computation is not likely to yield a large improvement in overall runtime for most datasets. Therefore, with no prior knowledge about the dataset, we recommend starting with the multi-run degree heuristic, forgoing the k-core computation. Then only if the run is OOM with this heuristic, would we recommend trying the multi-run core number version instead.

C. Windowing

Our goal with windowing was to reduce the number of candidates that need to be stored simultaneously, thereby reducing the memory requirements and allowing us to find the maximum clique for more datasets without running out of memory. We can see from the overall performance results in Figures 2 and 3 that using windowing generally decreases throughput, as we would expect when reducing the available parallel work. In this section, we look at how the choice of window size creates a tradeoff between this runtime increase and memory use reduction. We also test whether we can achieve any benefits from altering the order of the search by sorting the source vertices in the 2-clique list by their degrees.

1) Memory Use: **Windowing reduces the memory requirements by an average of 85-94%, with greater reductions for smaller window sizes. Searching the neighborhoods of more highly connected vertices first requires more memory than searching less connected vertices first or a random ordering.** For the regular breadth-first implementation, all candidates are stored until the search is complete, so memory use is only reduced by improving pruning. With windowing, memory requirements are determined by the largest subtree generated from a single window, which is affected by both pruning quality and window size. Pruning is affected by all factors discussed in previous sections, and can also be improved when a new best clique is found, increasing the lower bound for later windows. Pruning reduces work, but does not necessarily have a large effect on peak memory use, because we only need to store the clique lists generated from one window (as well as that of the best clique so far). We find that windowing improves pruning by an average of 20-26%, depending on the window size, but the reduction in memory usage is the most significant improvement.

Unsurprisingly, smaller window sizes provide larger memory savings, as shown in Figure 6. We find that by using windowing, we are able to solve 4 more (for a total of 56 out of 58 datasets) of the graph datasets that are OOM with the full breadth-first implementation. Sorting source vertices in descending degree order, thereby searching more highly connected vertices' neighborhoods first, uses more memory than searching in order of ascending degree or (randomized) index order. We might expect that prioritizing highly connected vertices would improve memory usage because the maximum clique(s) are more likely to contain these high-degree vertices, but we are also orienting the graph by degree, so larger cliques are more likely to be in low-degree vertices' candidate lists than they would be with index-based orientation. However, we also find that sorting the source vertices in ascending order does not significantly reduce peak memory usage over random order, suggesting that it is generally challenging to predict which sublist(s) the largest clique(s) are located in.

2) Runtime: **The smaller the window, the longer the runtime. Changing the traversal order does not have a significant effect on runtime.** We measure the speedups for the windowed version over the full breadth-first version and see a geometric mean speedup of $0.53x$ for a window size of 1024 and $0.89x$ for a window size of 32768. The runtime increases as the window size shrinks, which is to be expected, because we run the main loop on each window sequentially, so as the number of windows increases, so does the runtime. Additionally, depending on the number of candidates generated in the search, smaller windows may not provide enough parallelism to keep the GPU filled with work. Sorting source vertices did not have a significant effect on

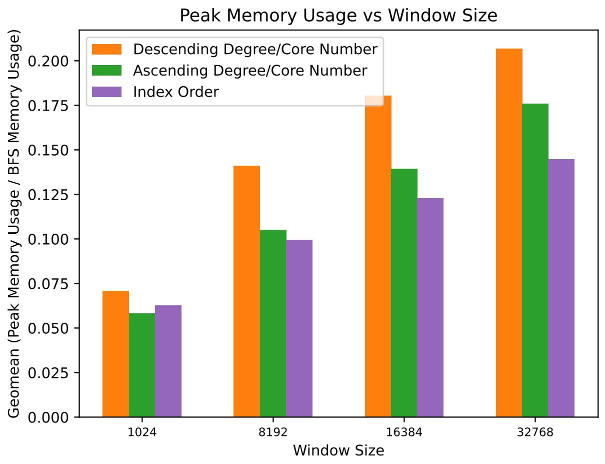


Fig. 6: Memory usage for windowed computation compared to full breadth-first maximum clique, using multi-run degree-based heuristic.

runtime. This suggests that performance is limited by the lack of available parallelism, since reducing the amount of work is not affecting the runtime. Overall, we find no memory or performance benefit in changing the order of search from a randomized order; however, depending on the default ordering of the graph dataset, it may be worthwhile to try sorting vertices in ascending degree order if needed to avoid OOM.

3) Recursive Windowing: Although we only implement windowing on the first level of the search, it is possible to perform windowing in later stages and/or multiple times during the search. The results shown in this section indicate that this would be an effective strategy for further reducing memory usage, but that the performance cost will also be quite high. Multiple windows could be explored simultaneously by different thread blocks in order to increase parallelism, but because the number of candidates generated by a window is unpredictable, managing the memory resources is challenging. Increasing windowing also moves the implementation further towards a regular depth-first implementation, which, as discussed in Section II-C, is not ideal for GPU performance.

VI. Conclusions

Memory requirements are the biggest constraint for our maximum clique implementation. Although there are a variety of strategies for improving pruning, we find that the fastest configuration is typically one that uses minimal preprocessing while simultaneously managing to avoid OOM. The goal is to choose a pruning strategy that is ”good enough” and not expend any further effort on pruning after that. This indicates that the breadth-first strategy was a good choice for the GPU, because indeed, the GPU performs well when it has lots of work, even if it could be ”easily” eliminated. However, BFS is not ideal for this problem in general, because (1) memory limits are easily reached with a combinatorial problem like this, and (2) the search will never finish early because what we are solving for is the depth itself.

VII. Acknowledgments

Thank you to Ryan Rossi for help understanding his work and the maximum clique problem more generally, and thanks to Adobe for initially funding this work with an Adobe Data Science Research Award. We also appreciate financial support from the National Science Foundation (Awards CCF-1637442 and OAC-1740333), the Department of Defense Advanced Research Projects Agency (Awards FA8650-18-2-7835 and HR0011-18-3-0007), and the U.S. Geological Survey (Award G21AP10845-00).

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin, “Parallel maximum clique algorithms with applications to network analysis,” *SIAM J. Scientific Computing*, vol. 37, no. 5, pp. C589–C616, Dec. 2015.
[2] J. Xiang, C. Guo, and A. Aboulnaga, “Scalable maximum clique computation using MapReduce,” in *29th IEEE International Conference on Data Engineering*, ser. ICDE 2013, April 2013, pp. 74–85.
[3] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, “Genome-scale computational approaches to memory-intensive applications in systems biology,” in *Proceedings of the ACM/IEEE Conference on Supercomputing*, ser. SC ’05, Nov 2005, p. 12.
[4] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, “Lessons learned from exploring the backtracking paradigm on the GPU,” in *Euro-Par 2011 Parallel Processing*. Springer Berlin Heidelberg, Aug. 2011, pp. 425–437.
[5] S. B. Seidman, “Network structure and minimum degree,” *Social Networks*, vol. 5, no. 3, pp. 269–287, Sep. 1983.
[6] 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, Apr. 2009.
[7] C. McCreesh and P. Prosser, “Multi-threading a state-of-the-art maximum clique algorithm,” *Algorithms*, vol. 6, no. 4, pp. 618–635, Oct. 2013.
[8] M. VanCompernolle, L. Barford, and F. Harris, “Maximum clique solver using bitsets on GPUs,” in *Information Technology: New Generations*. Springer International Publishing, Mar. 2016, pp. 327–337.
[9] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, “Maximal clique enumeration with data-parallel primitives,” in *IEEE 7th Symposium on Large Data Analysis and Visualization*, ser. LDAV ’17, Oct 2017, pp. 16–25.
[10] 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, Mar. 2021.
[11] J. W. Moon and L. Moser, “On cliques in graphs,” *Israel Journal of Mathematics*, vol. 3, no. 1, pp. 23–28, Mar. 1965.
[12] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, “Gunrock: GPU graph analytics,” *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.
[13] D. Merrill, “CUDA UnBound (CUB) library,” 2015–2022, https://nvlabs.github.io/cub/.
[14] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Jan. 2015, pp. 4292–4293.