Parallel Vertex Cover Algorithms on GPUs

Peter Yamout, Karim Barada, Adnan Jaljuli, Amer E. Mouawad, Izzat El Hajj

American University of Beirut, Lebanon

***Abstract*—Finding small vertex covers in a graph has applications in numerous domains such as scheduling, computational biology, telecommunication networks, artificial intelligence, social science, and many more. Two common formulations of the problem include: Minimum Vertex Cover (MVC), which finds the smallest vertex cover in a graph, and Parameterized Vertex Cover (PVC), which finds a vertex cover whose size is less than or equal to some parameter k. Algorithms for both formulations involve traversing a search tree, which grows exponentially with the size of the graph or the value of k.**

Parallelizing the traversal of the vertex cover search tree on GPUs is challenging for multiple reasons. First, the search tree is a narrow binary tree which makes it difficult to extract enough sub-trees to process in parallel to fully utilize the GPU's massively parallel execution resources. Second, the search tree is highly imbalanced which makes load balancing across a massive number of parallel GPU workers especially challenging. Third, keeping around all the intermediate state needed to traverse many sub-trees in parallel puts high pressure on the GPU's memory resources and may act as a limiting factor to parallelism.

To address these challenges, we propose an approach to traverse the vertex cover search tree in parallel using GPUs while handling dynamic load balancing. Each thread block traverses a different sub-tree using a local stack, however, we use a global worklist to balance the load to ensure that all blocks remain busy. Blocks contribute branches of their sub-trees to the global worklist on an as-needed basis, while blocks that finish their sub-trees pick up new ones from the global worklist. We use degree arrays to represent intermediate graphs so that the representation is compact in memory to avoid limiting parallelism, but self-contained which is necessary for the load balancing process.

Our evaluation shows that compared to approaches used in prior work, our hybrid approach of using local stacks and a global worklist substantially improves performance and reduces load imbalance, especially on difficult instances of the problem. Our implementations have been open sourced to enable further research on parallel solutions to the vertex cover problem and other similar problems involving parallel traversal of narrow and highly imbalanced search trees.

I. Introduction

A vertex cover of a graph is a set of vertices whose deletion from the graph (along with incident edges) induces an edgeless graph. Finding small vertex covers is one of the most famous problems in algorithmic graph theory and is among the original 21 NP-complete problems introduced by Karp in 1972 [1]. Finding small vertex covers has many applications in numerous domains such as scheduling, computational biology, telecommunication networks, artificial intelligence, social science, and many more [2], [3]. It is a problem that is particularly well-studied from the view point of parameterized complexity [4], [5], kernelization [6], [7], approximation [8], [9], exact exponential-time algorithms [10], [11], and heuristics [12], [13].

This work is supported by the Mamdouha El-Sayed Bobst Deanship Fund in the Faculty of Arts and Sciences at the American University of Beirut.

We consider two common formulations of the problem: Minimium Vertex Cover (MVC), which finds a vertex cover with the smallest number of vertices, and Parameterized Vertex Cover (PVC) which finds a vertex cover with $\leq k$ vertices for a given integer $k > 0$. Most algorithms for MVC and PVC traverse a binary tree to search for vertex covers, following the branch-and-reduce paradigm, also commonly known as branch-and-bound. At each node in the tree, reduction rules are first applied, followed by a check if a stopping criteria has been reached or if a solution has been found. If neither is the case, the tree branches into two sub-problems: one that removes the highest-degree vertex from the graph and adds it to the solution, and another that removes the neighbors of the highest-degree vertex from the graph and adds them to the solution.

Parallelizing the traversal of the vertex cover search tree on GPUs comes with many challenges. One challenge is extracting enough parallelism to fully utilize the GPU. Prior works [14], [15] divide the tree into sub-trees starting at the same depth and distribute these sub-trees across thread blocks. Since the search tree is narrow (binary), the sub-trees need to start at a deep level to ensure that enough parallelism is extracted. However, the deeper the starting level, the higher the overhead incurred for reaching these sub-trees due to redundancy [15] or grid launches and memory consumption [14]. Another challenge is dealing with load imbalance. Load imbalance is particularly challenging because the vertex cover search tree is highly imbalanced, so sub-trees have dramatically different sizes. Load imbalance on GPUs is typically addressed by extracting even more parallelism than the number of thread blocks that can run simultaneously to allow for dynamic load balancing. Indeed, prior work [15] extracts many more sub-trees than thread blocks for this reason. However, the imbalance in the vertex cover search tree is so high that one would need to start at very deep levels to ensure adequate load balancing, thus further increasing the overhead of reaching these sub-trees. A third challenge is ensuring that memory does not become a limiting factor for parallelism. As each thread block traverses a sub-tree, it needs to reserve a large amount of memory to maintain the intermediate traversal state. Hence, the memory capacity can limit the number of thread blocks that can execute in parallel.

One way to address these challenges is to use a global worklist that dynamically distributes work across thread blocks

on a per-tree-node instead of a per-sub-tree basis. However, such an approach would result in high contention on the queue and an exponential explosion in the number of queue entries. Instead, we propose a hybrid approach where each thread block uses a local stack to traverse a sub-tree, but contributes branches of its sub-tree to a global worklist as needed to ensure that there is enough work to keep all thread blocks busy. This hybrid approach extracts just enough parallelism to ensure load balance without incurring the overhead of redundancy or grid launches and memory capacity. We represent the intermediate graphs using degree arrays to ensure that the representation is compact so that memory consumption does not limit parallelism, but at the same time self-contained so that intermediate graphs can be shared across different thread blocks in the load balancing process.

We implement CUDA kernels for solving both MVC and PVC using our proposed approach and evaluate them on a server-grade GPU. Our evaluation shows that compared to the approach of distributing sub-trees starting at the same depth across thread blocks, our hybrid approach substantially improves performance and reduces load imbalance, especially on difficult instances of the problem and on graphs with a high average degree.

II. Background

A. Vertex Cover

We assume a graph $G = (V, E)$ that is finite, simple, and undirected [16]. The *neighborhood* of a vertex $v \in V(G)$ is the set of vertices adjacent to v, denoted by $N_G(v)$. The *degree* of a vertex $v \in V(G)$ is the number of edges incident on v, denoted by $d_G(v)$. The subscript G will be dropped when clear from the context. The maximum degree in G is denoted by $\Delta(G)$. Given a set of vertices $S \subseteq V(G)$, the subgraph induced by S is denoted by $G[S]$. $G - v$ and $G - S$ denote $G[V(G) \setminus \{v\}]$ and $G[V(G) \setminus S]$, respectively.

A set $S \subseteq V(G)$ is a *vertex cover* for G, if for every edge $uv \in E(G)$, we have $\{u, v\} \cap S \neq \emptyset$. Alternatively, one can view a vertex cover as a set of vertices whose deletion from the graph (along with incident edges) induces an edgeless graph. A vertex cover S is a *minimum vertex cover* for G if there is no vertex cover S' for G such that $|S'| < |S|$.

We consider two common formulations of the problem of finding small vertex covers in graphs. One formulation is Minimium Vertex Cover (MVC) which aims to find the minimum vertex cover S of G. Another formulation is Parameterized Vertex Cover (PVC) which, for a given integer $k > 0$, aims to find a vertex cover S of G such that $|S| \leq k$, if such a vertex cover exists. When k is larger than (or equal to) the size of a minimum vertex cover, PVC tends to be "faster" than MVC because the search terminates as soon as a solution of size k or less is found, in contrast with MVC where the search continues exploring the solution space to guarantee that no smaller solution exists.

```
1  Let best = APPROX_MVC(G)
2
3  function MVC(G, S)
4      (G, S) = reduce(G, S)
5      if |S| ≥ best ∨ |E(G)| > (best − |S| − 1)^2
6          // No MVC on this branch (do nothing)
7      else if |E(G)| == 0 // New MVC found
8          best = min(|S|, best)
9      else // Vertex cover not found, need to branch
10         Let v_max ∈ {u ∈ V(G) | d(u) = Δ(G)}
11         MVC(G − v_max, S ∪ {v_max})
12         MVC(G − N(v_max), S ∪ N(v_max))
13
14 function reduce(G, S)
15     do
16         graphHasChanged = false
17         while ∃v s.t. d(v) = 1
18             G = G − N(v)
19             S = S ∪ N(v)
20             graphHasChanged = true
21         while ∃v s.t. N(v) = {u, w} ∧ uw ∈ E(G)
22             G = G − N(v)
23             S = S ∪ {u, w}
24             graphHasChanged = true
25         while ∃v s.t. d(v) > best − |S| − 1
26             G = G − v
27             S = S ∪ {v}
28             graphHasChanged = true
29     while graphHasChanged
30     return (G, S)
```

Fig. 1. The serial algorithm for Minimum Vertex Cover. Initially $S = \emptyset$ and we assume that the graph has at least one edge, i.e., $|E(G)| \geq 1$.

B. Algorithms for Finding Vertex Covers

Most algorithms for MVC and PVC follow the well-known branch-and-reduce paradigm [11]. A *branch-and-reduce* algorithm searches the complete solution space of a given problem by *branching*, i.e., making decisions and solving smaller sub-problems. Due to the exponentially increasing number of potential solutions, the solution space is pruned using *reduction rules* derived from bounds on the function to be optimized and/or the value of the current best solution. At the implementation level, branch-and-reduce algorithms translate to search-tree-based algorithms. The search tree size usually grows exponentially with either the size of the input or, in the parameterized version, the value of the parameter k.

Figure 1 shows a branch-and-reduce algorithm for solving the MVC problem by traversing the vertex cover search tree. Figure 2 shows the tree traversed for an example graph (reduction rules are not applied in the example to keep the example small). The search tree is binary, branching into two sub-problems at every node in the tree. When visiting a node, the reduction rules are first applied (line 4, described later). Next, a stopping condition is checked to see if the search should stop or if it is still possible to find a better solution at this node or its descendants (line 5, described later). If it is possible, the algorithm checks if it has already arrived at a

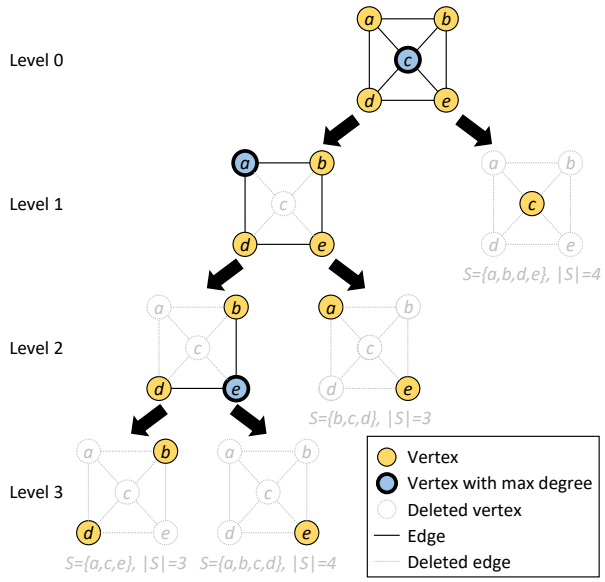

Fig. 2. Example of Vertex Cover Search Tree

vertex cover (line 7) and updates the best solution accordingly (line 8). If it has not arrived at a vertex cover, then it needs to branch. A vertex in the graph is selected as the basis for branching, which is typically the vertex of highest degree denoted by v_{max} (line 10). One branch removes v_{max} from the graph and adds it to the solution, while the other branch removes all the neighbors of v_{max} from the graph and adds them to the solution. The MVC function is called recursively to try and find a solution on each branch.

To initialize best (line 1), the minimum vertex cover is approximated using a greedy algorithm. The algorithm applies all reduction rules to the graph, removes the largest degree vertex from the graph (hence adding it to a solution), and repeats this process until a vertex cover is found.

The reduction rules applied are the *degree-one* reduction rule, the *degree-two-triangle* reduction rule, and the *high-degree* reduction rule. The rules are repeatedly applied until the graph stops changing. The degree-one reduction rule (lines 17-19) states that for any vertex v of degree one, either v or its neighbor u need to be in a solution to cover the edge uv. It is always at least as good to include u as to include v because u may have other incident edges that would also be covered. For the degree-two-triangle reduction rule (lines 21-23), if G contains a vertex v such that $N(v) = \{u, w\}$ and $uw \in E(G)$ (i.e., u, v, and w form a triangle), then two of the three vertices are needed to cover all the edges in the triangle. It is always at least as good to include u and w as to include only one of them and v because u and w may have other incident edges that would also be covered. Finally, the high-degree rule states that whenever a vertex v is found whose degree is greater than best$-|S|-1$, then adding all the neighbors of v to S can never achieve a solution better than best. Therefore, v is added to the solution.

The stopping condition (line 5) identifies if it is possible to find a solution at a node in the tree or its descendants. The condition deems it impossible in one of two sub-conditions. The first sub-condition is if the number of vertices added to a solution so far already exceeds best. The second sub-condition is based on the observation that the high-degree reduction rule has already removed all vertices with degree $>$ best$-|S|-1$. Hence, the remaining vertices have degree at most best$-|S|-1$. Moreover, finding a solution better than best would entail including no more than best $-\,|S|-1$ vertices. Hence, the maximum number of edges that can be covered to find such a solution is $(\text{best}-|S|-1)^2$. If the graph has more than that number of edges, then it is impossible to find a better solution on that branch.

As for the PVC problem, we omit the pseudocode for space constraints because it is largely similar to the pseudocode for MVC in Figure 1 with a few differences. For the high-degree reduction rule, $k-|S|$ is used instead of best $-\,|S|-1$ for comparison. For the stopping condition, the number of deleted vertices is compared to the parameter k instead of best, and the number of edges is compared to $(k-|S|)^2$ instead of $(\text{best}-|S|-1)^2$. When a vertex cover is found that does not exceed k, rather than updating best and continuing, the search is ended.

III. Challenges

Parallelizing the traversal of the vertex cover search tree on GPUs comes with numerous challenges. We discuss some of these challenges in this section and discuss how prior work has addressed them.

A. Challenge #1: Extracting Massive Parallelism

GPUs are massively parallel processors which require thousands of threads to fully utilize their computational resources. Hence, one must be able to extract many units of independent work from an application to parallelize the application on the GPU effectively. The typical way of extracting parallelism from a search tree traversal is by traversing independent sub-trees in parallel. For some graph applications that require search tree traversal [17], [18], the search tree is wide because the branching factor is large, which means that enough independent sub-trees can be extracted at the first or second level of the tree. However, the vertex cover search tree is a narrow binary tree, which means that enough parallelism is not available until deeper levels of the tree.

Prior work on accelerating vertex cover on GPUs [14], [15] consider a specific depth of the tree as the starting level, and treat all sub-trees starting at that level as independent units of parallelism. Sub-trees are distributed across thread blocks and each thread block traverses its sub-tree in a depth-first manner. This approach is illustrated in Figure 3. To reach the sub-trees, one approach [14] is to consecutively launch a separate grid for each level until the starting level is reached. However, this approach requires launching multiple grids and storing the state of all the initial sub-trees simultaneously. The deeper the starting level, the more the grid launches needed and the more the memory needed to store the state of the initial sub-trees.

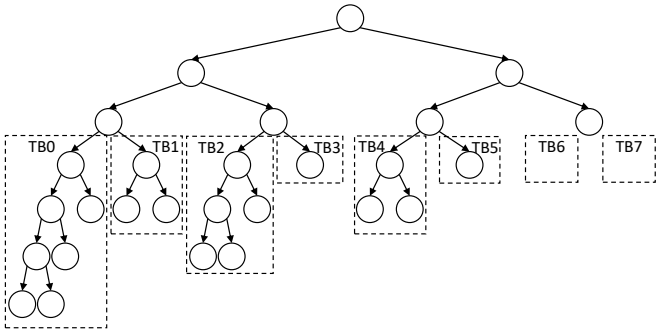

Fig. 3. Traversing Sub-trees in Parallel by Different Thread Blocks

Hence, there is a trade-off between the amount of parallelism extracted, and the grid launch and memory capacity overhead incurred by the extraction process. Another approach [15] is for each thread block to make its way down to its sub-tree from the root. However, this approach causes thread blocks assigned to nearby sub-trees to take redundant steps to arrive to their sub-trees. For example, TB0 and TB1 in Figure 3 both visit the same tree nodes in the first three levels. The deeper the starting level, the more the redundant work performed by different thread blocks. Hence, there is a trade-off between the amount of parallelism extracted and the amount of redundant work performed.

B. Challenge #2: Load Balancing

The massively parallel nature of GPUs makes them particularly sensitive to load imbalance. This sensitivity is especially challenging for the vertex cover problem. The vertex cover search tree is highly imbalanced because whenever the tree branches, one branch removes a single high-degree vertex from the graph whereas the other branch removes all the neighbors of that high-degree vertex; hence, the latter branch is likely to exceed the current minimum and terminate sooner. Because the search tree is imbalanced, prior work's approach of extracting parallelism via sub-trees starting at the same level leads to high load imbalance. For example, in Figure 3, TB0 receives a large sub-tree, TB5 only receives a single node, and TB7 does not even have a sub-tree to traverse.

One way to mitigate load imbalance on GPUs is to extract more units of independent work than the number of workers that can execute simultaneously such that there is enough work available for dynamic load balancing. For prior work, this would mean starting at a lower level in the search tree where there are many more sub-trees available than the number of thread blocks that can execute simultaneously. However, we have seen that starting at a deeper level in the tree could result in higher grid launch and memory overhead in one approach [14] or more redundant work in another approach [15].

C. Challenge #3: Memory as a Limiting Factor to Parallelism

GPUs have a relatively small memory capacity (compared to CPUs) while at the same time placing higher pressure on the memory capacity because of their massive parallelism. Memory capacity on the GPU can be a limiting factor for parallelism in two ways. First, the global memory capacity can limit the number of threads or thread blocks that can run simultaneously on the GPU if these threads or thread blocks require a large amount of global memory to store intermediate execution state. Second, the shared memory capacity per streaming multiprocessor (SM) can limit the occupancy of threads on the SM if a large amount of shared memory is required to store frequently accessed data. Both of these limits are encountered when traversing the vertex cover search tree.

For a thread block to traverse a sub-tree of the vertex cover search tree, it needs to manage a stack that stores the intermediate graph (a sub-graph with the solution vertices removed) at each level of the sub-tree. An explicitly managed stack is used instead of recursion because different threads in a block all need to access the same intermediate graph. Due to the expensive nature of dynamic memory allocation on GPUs, it is inefficient to grow the memory allocated for this stack dynamically. Instead, a stack for each thread block is pre-allocated in global memory and provisioned for the maximum possible depth that the tree can reach. Since GPUs execute many thread blocks at a time, enough memory needs to be available to support all the maximally-provisioned stacks for all the thread blocks simultaneously. These stacks grow with the size of the graph, which can make the global memory capacity a limiting factor for parallelism for large graphs.

When a thread block visits a node in its sub-tree, it frequently accesses the intermediate graph at that node. As an optimization, the intermediate graph can be placed in shared memory for fast access. However, placing the intermediate graph in shared memory can make the shared memory capacity a limiting factor for occupancy for large graphs. These global and shared memory capacity limits make efficient memory management an important challenge when parallelizing the traversal of the vertex cover search tree on GPUs.

IV. Parallelizing Vertex Cover on GPUs

A. Hybrid Traversal Approach using a Global Worklist

One way to address the challenges mentioned in Section III is by using a global worklist. Rather than assigning thread blocks to entire sub-trees, a thread block can be assigned to a single node in the tree. Upon branching, the thread block could add its node's children to a global worklist where other thread blocks can pick them up and process them. This approach substantially increases the amount of parallelism extracted from the traversal because it treats each tree node as a unit of parallelism as opposed to entire sub-trees. It also substantially reduces load imbalance because tree nodes are more similar to each other in load than entire sub-trees, and there is more of them to go around for dynamic load balancing. Additionally, a global worklist obviates the need for each thread block to maintain a local stack, which reduces the pressure on the global memory capacity. However, using a global worklist has two major drawbacks. The first drawback is that it converts the depth-first traversal of the search tree

into a breadth-first traversal, which results in an exponential explosion of the number of tree nodes that need to be added to the global worklist, quickly exceeding the worklist's capacity. The second drawback is that it creates a high amount of contention between thread blocks when accessing the global worklist, which becomes a serialization point in the program.

To reap the benefits of global worklists while mitigating their drawbacks, we propose a hybrid approach. In our approach, each thread block traverses a sub-tree in depth-first order while keeping track of its intermediate state using a local stack (stored in global memory). However, every time a thread block branches from a tree node, it first checks the global worklist. If the number of entries in the global worklist is below a certain threshold, the thread block will add one of its node's children to the global worklist and move to process the other child. Otherwise, if the number of entries in the global worklist exceeds the threshold, the thread block will push its node's child to its local stack and move to process the other child. When the thread block reaches the bottom of the tree and needs to find more work to do, it first attempts to pop a tree node from its local stack. If the local stack is empty, the thread block will then take a tree node from the global worklist and begin to traverse the sub-tree rooted at that node.

This hybrid approach of using both a global worklist and per-block local stacks captures the advantages of the two individual approaches. On the one hand, using a global worklist helps extract more parallelism from the computation and performs better dynamic load balancing. On the other hand, using the local stacks when the global worklist is sufficiently full prevents the exponential explosion of the number of tree nodes to be added to the global worklist. It also reduces contention on the global worklist since thread blocks will only add to the global worklist if it contains a small number of elements, and will always try to remove work from their local stacks before trying to remove from the global worklist.

Figure 4 shows the pseudocode for MINIMUM VERTEX COVER using our hybrid approach with a global worklist and per-block local stacks. Initially, all stacks are empty and the global worklist contains the root node of the tree. Each thread block first tries to pop work from its local stack (lines 5-6). If unsuccessful, the thread block tries to get work from the global worklist (lines 7-8). If the worklist indicates that the traversal is done (see Section IV-C), then the block terminates (lines 9-10). If the block is successful at obtaining a tree node from its local stack or the global worklist, then it starts by reducing the intermediate graph at that tree node (line 11). Next, it checks the stopping condition to see if it is still possible to find a solution on this branch (line 12, see Section II-B). If not possible, it sets a flag to obtain a new tree node from the stack or the global worklist on the next iteration. If it is still possible to find a solution, the block checks if it has already found a solution at the current tree node (line 17), and if yes, it atomically updates the current best solution size (line 18). It also sets a flag to obtain a new tree node from the stack or the global worklist on the next iteration (line 19). If the block has not found a solution yet then it needs to branch (line 20). The

```
1  function MVC(worklist, stack)
2      getNewTreeNodeNextItr = true
3      do
4          if getNewTreeNodeNextItr
5              if !stack.empty()
6                  (G, S) = stack.pop()
7              else
8                  ((G, S), done) = worklist.remove()
9                  if done
10                     break
11         (G, S) = reduce(G, S)
12         if |S| ≥ best ∨ |E(G)| > (best − |S| − 1)^2
13             // No MVC on this branch
14             getNewTreeNodeNextItr = true
15         else
16             Let v_max ∈ {u ∈ V(G) | d(u) = Δ(G)}
17             if d(v_max) == 0 // New MVC found
18                 best = min(best, |S|)
19                 getNewTreeNodeNextItr = true
20             else // Need to branch
21                 G' = G − N(v_max)
22                 S' = S ∪ N(v_max)
23                 if worklist.numEntries ≥ threshold
24                     stack.push(G', S')
25                 else
26                     worklist.add(G', S')
27                 G = G − {v_max}
28                 S = S ∪ {v_max}
29                 getNewTreeNodeNextItr = false
30     while(true)
```

Fig. 4. Minimum Vertex Cover using a Hybrid Approach with a Global Worklist and Per-block Local Stacks

block sets up one of the child nodes by removing the neighbors of the max-degree vertex from the graph and adding them to the vertex cover (lines 21-22). If the worklist is sufficiently full (line 23), the child is pushed to the stack (line 24), otherwise it is added to the worklist (lines 25-26). The thread block then sets up the other child node by removing just the max-degree vertex from the graph and adding it to the vertex cover (lines 27-28). The block will proceed to process this child node on the next iteration, so it sets a flag that it does not need to obtain a new tree node from the stack or the global worklist on the next iteration.

For space constraints, we omit the pseudocode for PVC which is largely similar to that of MVC in Figure 4 with a few differences. The differences in the stopping condition are already described in Section II-B. Another difference is that when we find a vertex cover whose size does not exceed k, rather than updating best and continuing, we set a flag telling other blocks that a vertex cover has been found and we terminate. We also add a condition at the beginning of the loop (before line 4) where blocks check the flag and terminate if a vertex cover has been found before picking up another tree node to work on.

With this hybrid approach, we have tackled the first two challenges described in Section III, namely extracting massive

parallelism and load imbalance. However, we are still using a local stack per thread block so the third challenge of memory as a limiting factor to parallelism remains. We address this challenge in upcoming sections.

B. Graph Representation and Operations

The original graph is represented using the commonly used Compressed Sparse Row (CSR) format [19]. This representation is compact, requiring $\mathcal{O}(|V|+|E|)$ memory. It also makes it easy to access the incident edges and neighbors of a given vertex. A single copy of the CSR graph representation exists and is accessed by all thread blocks and never modified.

The intermediate graphs which are stored in the local stacks and the global worklist are not represented using CSR. One reason is that the CSR format is expensive to modify which makes it inconvenient for removing vertices and edges from the graph. Another reason is that the CSR format would make the local stacks and global worklist require too much memory, even with $\mathcal{O}(|V| + |E|)$ memory consumption. Recall from Section III-C that efficient memory management is critical to be able to support large graphs.

Instead of using CSR, the intermediate graphs along with the set of removed vertices ((G, S) in Figure 4) are jointly represented using just a *degree array*. The degree array is an array with one element per original vertex that stores the degree of the vertex if the vertex is still in the graph, or a sentinel value if the vertex has been removed from the graph and added to the solution $|S|$. The degree array representation has been used to represent intermediate graphs when searching for vertex covers [14], [20]. It is particularly useful in our implementation for two reasons. The first reason is that the array only consumes $\mathcal{O}(|V|)$ memory which limits the memory consumption of the local stacks and global worklist. The second reason is that when combined with the original graph, it is sufficient to represent the updated graph without any other information. This property is important for dynamic load balancing because we need to be able to put (G, S) in the global worklist where any other thread block can pick it up, so (G, S) must be self-contained.

Operations on the intermediate graph in Figure 4 are performed in parallel via collaboration between the threads within the block. Applying the reduction rules (line 11) is discussed separately in Section IV-D. To find the vertex with maximum degree (line 16), a parallel reduction tree is performed on the degree array. To remove a single vertex from the graph and add it to the solution (lines 27-28), the vertex's degree is set to a sentinel value by one thread and the vertex's neighbors are distributed across the threads to decrement their degrees in parallel. To remove the neighbors of a single vertex from the graph and add them to the solution (lines 21-22), the vertex's neighbors are distributed across the threads. For each neighbor, a thread will iterate over the neighbor's neighbors and atomically decrement their degrees, then set the degree of the neighbor to a sentinel value. To find the number of vertices in the solution ($|S|$ on lines 12 and 18), a reduction tree could be performed over the degree array to count the number of sentinel values. However, as an optimization, we store an additional counter with the degree array that tracks the number of deleted vertices, and update that counter whenever we delete a vertex.

C. Implementation of the Global Worklist

We implement the global worklist using the Broker Work Distributor (BWD) [21] which is a state-of-the-art worklist data structure for dynamic work distribution. We make one modification to the BWD data structure to support our algorithm. By design, if the worklist is empty, BWD returns that it cannot remove any elements from the worklist. The worklist is considered empty if all blocks that have added or committed to add an entry have corresponding blocks that have committed to remove an entry. However, the worklist may be empty in one of two situations. The first situation is where some blocks are still executing and may commit to add entries to the worklist in the future. In this situation, we would like to keep checking the worklist until the new work arrives. The second situation is where no blocks are executing and they are all trying to remove work from the empty worklist. In this situation, we can expect that no blocks will commit to add work in the future, which means that the work is done and we can safely terminate.

To handle these two situations, we wrap the BWD function for removing worklist entries in a loop. Each iteration of the loop first attempts to remove an entry from the worklist. If it succeeds, we return this entry so the block can process it. If it fails, we atomically check if the worklist is empty and if the number of thread blocks trying to remove from the worklist is all the thread blocks in the grid. If the check succeeds, we return that we are done so the block can exit (lines 9-10 in Figure 4). In the parameterized version, we also check the flag that indicates that a vertex cover has been found. If the flag is set, then we return that we are done. If we are not done, then we let the thread block sleep for some time then go back to the beginning of the loop and repeat the process.

D. Reduction Rules

When a thread block visits a tree node, it applies the three reduction rules (degree-one, degree-two-triangle, and high-degree) described in Section II-B until the graph no longer changes. Each rule is executed in parallel by all the threads in the block, so care must be exercised when implementing them in parallel. For the degree-one rule, different threads find different degree-one vertices simultaneously. However, different degree-one vertices may have a common neighbor so care is exercised to ensure that the neighbor is removed only once. Moreover, two threads may simultaneously find two degree-one vertices that are neighbors of each others, so only one of the two vertices is removed (the one with the smaller vertex ID), not both. For the degree-two-triangle rule, different threads find different degree-two vertices simultaneously and check if they are part of a triangle. However, different degree-two vertices may participate in the same triangle, so the neighbors of only one of these vertices (the one with the

smaller vertex ID) are removed. We handle all these cases in our parallel implementation of the reduction rules.

E. Memory Management

Recall from Section III-C that efficient memory management is critical for supporting large graphs. The total amount of global memory needed for storing all the per-block local stacks is dependent on three factors: the size of a stack entry (i.e., size of the intermediate graph), the number of stack entries per stack (i.e., maximum depth of the search tree), and the number of stacks (i.e., number of thread blocks). We have already seen in Section IV-B that we limit the size of a stack entry by representing the intermediate graph using a degree array which requires $\mathcal{O}(|V|)$ space. To limit the number of stack entries per stack, we run the greedy algorithm to approximate the minimum vertex cover on the CPU (see Section II-B) and use the approximation as the limit on the stack depth when starting the GPU kernel since no thread block will ever go deeper in the tree than the size of this minimum. In the parameterized version, the parameter k is used as the bound. To limit the number of stacks, we must limit the number of thread blocks as the size of the graph gets larger. To limit the number of thread blocks while maintaining the total number of threads needed to achieve the highest device occupancy, we must use a large number of threads per block. Hence, the number of threads per block must be carefully selected based on the size of the graph to ensure that the highest device occupancy is achieved for large graphs.

As for shared memory, shared memory is primarily used for each thread block to store the intermediate graph for the tree node it is currently working on. The total amount of shared memory needed per SM depends on two factors: the amount of shared memory needed per block (i.e., the size of the intermediate graph) and the number of blocks per SM. We have already seen that we limit the size of the intermediate graph by using a degree array to represent it which requires $\mathcal{O}(|V|)$ space. To limit the number of thread blocks per SM while maintaining the total threads needed to achieve maximum SM occupancy, we must use a large number of threads per block. Hence, the choice of the number of threads per block not only considers the impact of the global memory capacity on the number of blocks that can run concurrently on the device, but also the impact of the shared memory capacity on the number of blocks that can run concurrently per SM.

To satisfy these constraints while maximizing occupancy, we select then number of threads per block as follows. We determine an upper-limit on the number of threads per block based on the hardware limit on the block size and $|V(G)|$, whichever is smaller. We use $|V(G)|$ as an upper-limit because it is not useful to have more threads in the block than the number of vertices in the graph because these threads will not perform any work. We also determine a lower-limit on the number of threads per block based on the desired number of threads to achieve full occupancy and the upper-limit on number of blocks that can run simultaneously. The upper-limit on the number of blocks is the minimum of the following limits: the hardware limit on the number of simultaneous blocks, the shared memory limit on the number of simultaneous blocks, and the global memory limit on number of simultaneous blocks (i.e., number of stacks that can be stored). If the lower-limit is less than the upper-limit, we select a thread block size within the range that is a power of two. If the lower-limit is greater than the upper-limit, then it is impossible to achieve full occupancy. In this case, we select the upper-limit as the thread block size and let the kernel execute without achieving full occupancy. In practice, the shared memory capacity tends to be more restrictive than the global memory capacity for most graphs. For this reason, we provide two versions of each kernel, one that uses shared memory to store the intermediate graph that the block is currently working on, and one that uses global memory to store the intermediate graph. If the lower-limit is too high because of the shared memory constraint, we relax the shared memory constraint by falling back on the kernel that uses global memory to store the intermediate graph.

V. Evaluation

A. Methodology

We implement and evaluate three different code versions:

- *Sequential*: This implementation executes on a single CPU thread. The objective of evaluating this implementation is just for reference. A fair comparison to CPUs would entail comparing to a parallel CPU implementation, but this is not the aim of our work. Our work aims to show how the vertex cover search tree traversal can be parallelized using GPUs.
- *StackOnly*: This implementation parallelizes sub-trees starting at a specific level across thread blocks. Each thread block makes its way down to its sub-tree from the root then proceeds to traverse its sub-tree using a per-block local stack, similar to what prior work does [15].
- *Hybrid*: This implementation uses the hybrid approach that leverages per-block local stacks as well as a global worklist to assist with dynamic load balancing.

For a fair comparison, all the versions use the same data structure, apply the same reduction rules, and use the same strategy to compute an approximate minimum on the CPU before traversing the search tree.

We implement our code using C++ and CUDA. The CPU implementation is evaluated on an AMD EPYC 7551P CPU with 64GB of main memory. The GPU implementations are evaluated on a Volta V100 GPU with 32GB of device memory.

For the *StackOnly* and *Hybrid* implementations, we follow the strategy described in Section IV-E to select between the shared memory and global memory kernels and to select the number of threads per block (block size). The shared memory kernel is selected for all high-degree graphs (small $|V|$) and the global memory kernel is selected for all low-degree graphs (large $|V|$). If multiple block sizes are possible, we try them all and report the best result. However, one can still obtain performance benefits without selecting the best

TABLE I
EXECUTION TIME (IN SECONDS)

	Graph	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	MVC			PVC								
								k = min − 1			*k = min*			*k = min + 1*		
					Sequential	StackOnly	Hybrid	Sequential	StackOnly	Hybrid	Sequential	StackOnly	Hybrid	Sequential	StackOnly	Hybrid
High degree	p_hat_300_1 [22]	300	33917	113	0.138	0.780	**0.021**	0.140	0.782	**0.023**	0.031	0.021	**0.016**	0.028	**0.001**	0.003
	p_hat_300_2 [22]	300	22922	76	1.262	15.681	**0.029**	1.266	15.714	**0.029**	0.016	0.021	**0.016**	0.016	**0.010**	0.016
	p_hat_300_3 [22]	300	11460	38	200.990	2,197.583	**1.657**	193.597	2,199.056	**1.658**	**0.047**	0.528	0.056	0.006	**0.005**	0.008
	p_hat_500_1 [22]	500	93181	186	1.150	7.787	**0.092**	1.456	7.823	**0.090**	0.146	0.145	**0.019**	0.139	**0.006**	0.007
	p_hat_500_2 [22]	500	61804	124	102.553	1,541.506	**1.558**	100.602	1,542.344	**1.559**	**0.069**	0.122	0.101	0.072	**0.036**	0.070
	p_hat_500_3 [22]	500	30950	62	> 2 hrs	> 2 hrs	**1,018.898**	> 2 hrs	> 2 hrs	**1,027.504**	**2.480**	928.941	25.636	**0.022**	0.083	0.095
	p_hat_700_1 [22]	700	183651	262	4.838	31.245	**0.238**	8.054	31.200	**0.178**	0.672	0.584	**0.188**	0.409	0.593	**0.075**
	p_hat_700_2 [22]	700	122922	176	1,949.591	> 2 hrs	**31.241**	1,833.827	> 2 hrs	**31.507**	2.903	42.947	**0.243**	0.221	**0.060**	0.074
	p_hat_1000_1 [22]	1000	377247	377	58.056	495.296	**1.400**	63.104	495.099	**1.397**	1.456	5.099	**0.135**	1.151	0.043	**0.017**
	p_hat_1000_2 [22]	1000	254701	255	> 2 hrs	> 2 hrs	**4,527.601**	> 2 hrs	> 2 hrs	**4,596.877**	**1.263**	8.128	4.099	**0.627**	0.902	0.939
	movielens-100k_rating [23]	2625	94834	36	4.906	**0.115**	0.132	4.840	**0.114**	0.133	**0.012**	0.019	0.023	**0.012**	0.019	0.023
	wikipedia_link_lo [23]	3811	83029	22	> 2 hrs	> 2 hrs	**387.628**	> 2 hrs	> 2 hrs	**421.803**	> 2 hrs	**0.031**	0.045	**0.020**	0.030	0.047
	wikipedia_link_csb [23]	5561	187269	34	0.372	39.227	**0.034**	0.151	39.147	**0.035**	0.158	0.007	**0.007**	0.107	0.007	**0.006**
Low degree	US power grid [23]	4942	6594	1.33	145.574	1.518	**0.852**	141.734	1.531	**0.853**	0.002	0.001	**0.001**	0.002	0.001	**0.001**
	LastFM Asia [24]	7624	27806	3.65	83.389	4.345	**0.939**	81.894	4.395	**1.052**	0.005	0.009	**0.005**	0.005	0.008	**0.005**
	Sister Cities [23]	14275	20573	1.44	5.634	2.850	**0.106**	5.526	2.853	**0.116**	0.004	0.005	**0.002**	0.004	0.005	**0.003**
	vc-exact_023 [25]	27718	133665	4.82	> 2 hrs	> 2 hrs	>2 hrs	> 2 hrs	> 2 hrs	> 2 hrs	1.539	0.898	**0.878**	1.537	0.898	**0.881**
	vc-exact_009 [25]	38453	174645	4.54	> 2 hrs	> 2 hrs	> 2 hrs	> 2 hrs	> 2 hrs	> 2 hrs	2.883	**1.605**	1.651	2.878	**1.605**	1.642

block size. Sub-optimal selection of the block size would cause a geometric mean slowdown of 1.55× in the average case and 2.40× in the worst case for the StackOnly implementation, and 1.39× in the average case and 1.80× in the worst case for the Hybrid implementation. Hence, the Hybrid implementation is more robust than the StackOnly implementation to a sub-optimal selection of block size, and the slowdown of a sub-optimal selection is within the speedup margins reported in our evaluation.

For the StackOnly implementation, to select the starting depth, we try three different depth values (8, 12, and 16) and report the best result. Sub-optimal selection of the starting depth would result in a geometric mean slowdown of 1.18× in the average case and 1.37× in the worst case.

For the Hybrid implementation, we try global worklist sizes of $128K$, $256K$, and $512K$ entries, and threshold values of 0.25×, 0.5×, 0.75×, and 1.0× the worklist size and report the best result. Sub-optimal selection of the worklist size and threshold would result in a geometric mean slowdown of 1.18× in the average case and 1.32× in the worst case, which is within the speedup margins reported in our evaluation.

B. Performance

Table I shows the execution time of each implementation for four different instances of the problem (MVC, PVC with $k = min-1$, PVC with $k = min$, PVC with $k = min+1$) across a wide range of graphs obtained from popular collections [22]–[25]. We take the edge complements of graphs in the DIMACS collection [22] like in prior work [15]. Table II shows the speedup of the Hybrid implementation over the StackOnly implementation and the CPU implementation (Sequential) for the four instances, aggregated across two categories of graphs: graphs with high average degree (denoted as high-degree) and graphs with low average degree (denoted as low-degree). Based on these results, we make three key observations.

The first observation is that the Hybrid implementation substantially outperforms the StackOnly implementation on high-degree graphs, while having moderate performance advantage on low-degree graphs. Recall from Section III-B that the vertex cover search tree is imbalanced because whenever the tree branches, one branch removes a single high-degree vertex from the graph whereas the other branch removes all the neighbors

TABLE II
AGGREGATE SPEEDUP (GEOMETRIC MEAN)

Category	Speedup of Hybrid over StackOnly				Speedup of Hybrid over Sequential			
	MVC	PVC			MVC	PVC		
		k=min−1	*k=min*	*k=min+1*		*k=min−1*	*k=min*	*k=min+1*
High-degree	167.1×	171.3×	4.2×	0.9×	30.0×	30.1×	1.8×	2.4×
Low-degree	6.1×	5.7×	1.2×	1.2×	93.1×	85.0×	1.5×	1.5×
Overall	72.9×	73.1×	3.0×	1.0×	39.0×	38.2×	1.7×	2.1×

of that high-degree vertex. The higher the average degree of the graph, the higher the disparity in how many vertices are removed by each branch on average, and therefore, the more imbalanced the search tree is likely to be. Since high-degree graphs are likely to have more imbalanced search trees, they are likely to benefit more from the load balancing that the Hybrid implementation provides.

The second observation is that the Hybrid implementation substantially outperforms the StackOnly implementation on the difficult instances with long run-times (MVC and PVC with $k = min - 1$), while having comparable performance on the easier instances with short run-times (PVC with $k = min$ and $k = min + 1$). PVC with $k = min$ and PVC with $k = min + 1$ stop as soon as a solution is found on one branch of their search trees, whereas MVC searches all branches of its tree to find the smallest vertex cover and PVC with $k = min - 1$ searches all branches of its search tree without finding any solution. Because MVC and PVC with $k = min - 1$ search their trees more exhaustively, they are more likely to run into deeper branches that cause load imbalance, which makes them more likely to benefit from the load balancing that the Hybrid implementation provides.

We look further into the first and second observation in Section V-C where we analyze the load balance of each implementation more thoroughly on different graphs and for different instances. As shown in Table II, the Hybrid implementation is faster than the StackOnly implementation on high-degree graphs by 167.1× for MVC and 171.3× for PVC with $k = min - 1$. Although the StackOnly implementation does outperform the Hybrid implementation in a few instances for select graphs as shown in Table I, these cases tend to be on easier instances with short run-times and the performance difference is not usually significant. For this reason, we are not motivated to design a criteria for selecting between the

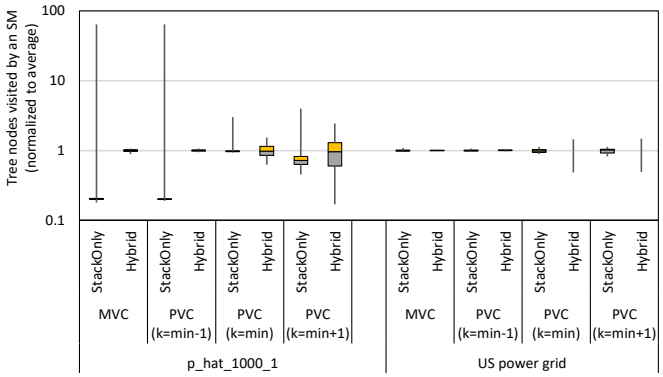

Fig. 5. Distribution of Load across SMs

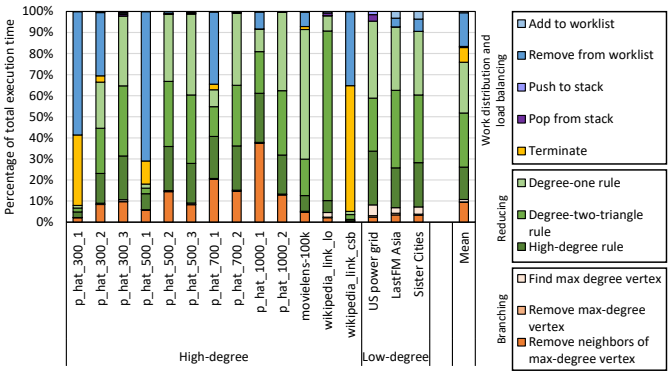

Fig. 6. Breakdown of Execution Time for MVC

two GPU implementations.

Our third observation is that the Hybrid GPU implementation outperforms the Sequential CPU implementation substantially, especially for difficult instances with long run-times (MVC and PVC with $k = min - 1$). As mentioned in Section V-A, a fair comparison to CPUs would entail comparing to a parallel CPU implementation. We compare to Sequential just for reference to show that GPUs can have competitive performance compared to CPUs in this tree traversal algorithm that is normally considered difficult to parallelize.

C. Load Balance

Figure 5 compares the load distribution achieved by each of the StackOnly and Hybrid implementations for the four instances of the problem on two sample graphs. Load is measured as the ratio of the number of tree nodes visited by an SM to the average number of tree nodes visited across all SMs. The graphs picked are those at the two extremes, having the highest average degree (p_hat_1000_1) and the lowest average degree (US power grid). We make three key observations.

The first observation is that the StackOnly implementation has substantially higher load imbalance on the high-degree graph than on the low-degree graph. The second observation is that the StackOnly implementation has substantially higher load imbalance on the difficult instances with long run-times (MVC and PVC with $k = min-1$) than on the easier instances with short run-times (PVC with $k = min$ and $k = min + 1$). These observations are consistent with the points mentioned in Section V-B that high-degree graphs and difficult long-running instances are likely to suffer from more load imbalance.

The third observation is that the Hybrid implementation achieves better load balance than the StackOnly implementation. For example, when the StackOnly implementation solves MVC on p_hat_1000_1, more than 75% of the SMs take less than 0.21× the average load, whereas one SM takes 63.98× the average load. In contrast, when the Hybrid implementation solves the same instance, the least loaded SM takes 0.89× the average load whereas the most loaded SM takes 1.07× the average load. These results demonstrate the effectiveness of the Hybrid implementation at achieving load balance.

D. Breakdown of Execution Time

To better understand where our traversal code is spending time, Figure 6 shows the breakdown of the execution time of our MVC kernel for different graphs. To measure this breakdown, we instrument our code to use the SM clocks to get the number of cycles spent by each thread block on each activity. We then normalize the cycle counts to the total number of cycles executed by the thread block and take the mean across all thread blocks. We make three key observations.

The first observation is that the kernel spends 24.1% of its time on average on activities related to work distribution and load balancing, of which 16.0% is spent removing from the global worklist. Removing from the worklist is expensive because there may be high contention on the worklist from other blocks, or because the worklist may be empty requiring the block to wait for new work to arrive (see Section IV-C). This overhead is acceptable given the difficulty of extracting parallelism and load balancing in this particular problem.

The second observation is that the kernel spends 65.2% of its time (the majority of its time) on average on the reduction rules. This time is well-spent because the reduction rules allow the kernel to make the fastest progress towards finding a solution. The time is distributed almost evenly across the different rules.

The third observation is that the kernel spends 10.7% of its time on average on activities related to branching, of which 9.4% is spent on removing the neighbors of the max-degree vertex. It is noteworthy that removing the neighbors of the max-degree vertex takes less time in low-degree graphs than in high-degree graphs since there are fewer neighbors to remove on average in the low-degree graphs.

E. Comparison with Prior Work

Table III compares the performance of our implementation to that of the most recent prior work for GPUs [15]. This prior work uses an approach similar to our StackOnly approach which distributes sub-trees starting at a specific level across thread blocks then has each block make its way down to its sub-tree from the root and traverse its sub-tree using a per-block local stack. In particular, it solves the PVC instance and

TABLE III
COMPARISON OF EXECUTION TIME (IN SECONDS) WITH PRIOR WORK

Graph	Sequential	StackOnly	Hybrid	Abu Khuzam et al. [15]
p_hat_300_1	0.031	0.021	**0.016**	4.400
p_hat_300_2	0.016	0.021	**0.016**	5.000
p_hat_300_3	**0.047**	0.528	0.056	2.800
p_hat_500_1	0.146	0.145	**0.019**	10.700
p_hat_500_2	**0.069**	0.122	0.101	10.100
p_hat_500_3	**2.480**	928.941	25.636	6.000
p_hat_700_1	0.672	0.584	**0.188**	21.000
p_hat_700_2	2.903	42.947	**0.243**	14.800
p_hat_1000_1	1.456	5.099	**0.135**	48.300
p_hat_1000_2	**1.263**	8.128	4.099	30.800

evaluates using $k = min$. The times reported in the paper [15] are used directly for comparison and are replicated in Table III. We note that this comparison is not fair because prior work uses two AMD FirePro D500 GPUs with 3GB of memory each, while we use a more powerful Volta V100 GPU with 32GB of memory. However, we are unable to evaluate their performance on our system because the code is not publicly available. The objective of this comparison is to show that our approach is highly competitive with prior GPU solutions for the vertex cover problem.

VI. RELATED WORK

In the last few decades, a lot of effort has been devoted to developing fast and simple exact algorithms for NP-hard problems [26] and MVC is no exception. One of the first examples is the $\mathcal{O}(2^{n/3})$-time algorithm of Tarjan and Trojanowski [27] for MAXIMUM INDEPENDENT SET (MIS) on n-vertex graphs. Note that MIS is equivalent to MVC since the complement of a minimum vertex cover is a *maximum independent set*, i.e., a maximum set of pairwise non-adjacent vertices. The aforementioned $\mathcal{O}(2^{n/3})$-time algorithm is significantly faster than the trivial $\mathcal{O}(2^n)$-time brute-force algorithm. Considerable improvements were made in the algorithm of Robson [28], [29] which runs in $\mathcal{O}(1.1889^n)$-time (further improvements are also known [30], [31]).

For PVC, when the size of the vertex cover we are looking for, denoted by k, is sufficiently smaller than n, much faster algorithms exist. A problem is said to be *fixed-parameter tractable (FPT)*, if it can be solved in time $f(k) \cdot n^{\mathcal{O}(1)}$, where f only depends on k (usually exponential in k) and n is the size of input. In 1988, Fellows provided an $\mathcal{O}(2^k \cdot n)$ algorithm for PVC, showing that the problem is fixed-parameter tractable (a recent exposition can be found in Downey and Fellows [32]). The algorithm is based on the bounded search tree technique discussed in Section II-B. Since then, and after a long series of works [5], [33]–[36], the asymptotic upper bound on the running time of PVC was improved to $\mathcal{O}(1.2738^k + kn)$ by Chen et al. [4].

There are many serial [37]–[39] and parallel [40], [41] implementations that solve the vertex cover problem on CPUs. Our work focuses on solving the problem on GPUs, which has only recently gained attention. Some recent works provide approximate/heuristic algorithms for MVC [42] and MIS [43], [44] on GPUs. The focus of our work is on the exact algorithms which follow the hard-to-parallelize branch-and-reduce paradigm. Section III already compares to prior works [14], [15] that parallelize exact vertex cover algorithms on GPUs. These works distribute sub-trees starting at at the same level across thread blocks. We show that our approach can achieve substantially better performance via improved load balancing. Liu et al. [45] traverse the top of the tree on the CPU and send sub-trees to the GPU whenever the size of the graph drops below a certain threshold. This approach requires frequent communication between the CPU and the GPU, and results in launching many small grids (one single-block grid per sub-tree) which is known to underutilize device resources.

Search tree traversal on GPUs has been explored in the context of other problems. For example, recent work has been done on graph pattern mining [46], [47], maximal clique enumeration [17], [48], and k-clique counting [18]. These problems usually have a sufficient number of sub-trees available at the first or second level of the search tree such that distributing sub-trees across thread blocks can achieve adequate load balance. The vertex cover problem is different in that the search tree is narrower and highly imbalanced, which makes extracting enough parallelism more difficult. Other problems involving search tree traversal that have been solved on GPUs include the N-Queens problem [49] and minimax tree search [50]. To extract enough parallelism, these approaches distribute sub-trees across threads or blocks starting at a certain depth in the tree, similar to what prior work [14], [15] does for the vertex cover problem. Our work uses a global worklist to allow thread blocks to contribute branches of their sub-trees at any level for other idle blocks to process. Another work on the N-Queens problem [51] uses dynamic parallelism to parallelize the search tree traversal. We avoid using dynamic parallelism in our implementation because it is known to be inefficient when many small grids are launched [52], [53].

VII. CONCLUSION

We present techniques for parallelizing exact algorithms for MINIMUM VERTEX COVER and PARAMETERIZED VERTEX COVER on GPUs. We propose a hybrid approach for work distribution and dynamic load balancing where each thread block uses a local stack to traverse a sub-tree, but contributes branches of its sub-tree to a global worklist on an as-needed basis, extracting just enough parallelism for load balancing without incurring too much overhead. We represent intermediate graphs using degree arrays to ensure that they are compact so that memory consumption does not limit parallelism, but at the same time self-contained so that they can be shared across different thread blocks in the load balancing process. We implement CUDA kernels for solving both MVC and PVC using our proposed approach, and show that they achieve substantial performance and load balance improvements, especially on difficult instances of the problem and on graphs with high average degree. Our implementations have been open sourced to enable further research on parallel solutions to the vertex cover problem and other similar problems involving parallel traversal of narrow and highly imbalanced search trees.

REFERENCES

[1] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a Symposium on the Complexity of Computer Computations*, R. E. Miller *et al.*, Eds. Plenum Press, New York, 1972, pp. 85–103.

[2] S. Bhattacharyya *et al.*, "Resynchronization for multiprocessor DSP systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, no. 11, pp. 1597–1609, 2000.

[3] D. Vigo *et al.*, "Modeling and solving the crew rostering problem," *Operations Research*, vol. 46, 11 1996.

[4] J. Chen *et al.*, "Improved upper bounds for vertex cover," *Theoretical Computer Science*, vol. 411, no. 40-42, pp. 3736–3756, 2010.

[5] R. Balasubramanian *et al.*, "An improved fixed-parameter algorithm for vertex cover," *Information Processing Letters*, vol. 65, no. 3, pp. 163–168, 1998.

[6] M. R. Fellows *et al.*, "What is known about vertex cover kernelization?" in *Adventures Between Lower Bounds and Higher Altitudes*, H. Böckenhauer *et al.*, Eds., vol. 11011. Springer, 2018, pp. 330–356.

[7] J. Chen, "Vertex cover kernelization," in *Encyclopedia of Algorithms - 2008 Edition*, M. Kao, Ed. Springer, 2008.

[8] G. Karakostas, "A better approximation ratio for the vertex cover problem," *ACM Transactions on Algorithms*, vol. 5, no. 4, pp. 41:1–41:8, 2009.

[9] F. Delbot *et al.*, "New approximation algorithms for the vertex cover problem," in *24th International Workshop Combinatorial Algorithms IWOCA*, vol. 8288. Springer, 2013, pp. 438–442.

[10] F. Grandoni, "Exact algorithms for maximum independent set," in *Encyclopedia of Algorithms*, 2016, pp. 680–683.

[11] G. J. Woeginger, "Exact algorithms for NP-hard problems: A survey," in *Combinatorial Optimization - Eureka, You Shrink!*, M. Jünger *et al.*, Eds., vol. 2570. Springer, 2001, pp. 185–208.

[12] I. K. Evans, "Evolutionary algorithms for vertex cover," in *7th International Conference Evolutionary Programming*, vol. 1447. Springer, 1998, pp. 377–386.

[13] S. Voß *et al.*, "A hybridized tabu search approach for the minimum weight vertex cover problem," *J. Heuristics*, vol. 18, no. 6, pp. 869–876, 2012.

[14] R. K. Kabbara, "A parallel search tree algorithm for vertex cover on graphical processing units," Master's thesis, Lebanese American University, 2013.

[15] F. N. Abu-Khzam *et al.*, "Accelerating vertex cover optimization on a GPU architecture," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 616–625.

[16] R. Diestel, *Graph Theory, 4th Edition*, ser. Graduate texts in mathematics. Springer, 2012, vol. 173.

[17] J. Jenkins *et al.*, "Lessons learned from exploring the backtracking paradigm on the GPU," in *European Conference on Parallel Processing*. Springer, 2011, pp. 425–437.

[18] M. Almasri *et al.*, "K-clique counting on GPUs," *arXiv preprint arXiv:2104.13209*, 2021.

[19] N. Bell *et al.*, "Efficient sparse matrix-vector multiplication on CUDA," Citeseer, Tech. Rep., 2008.

[20] F. N. Abu-Khzam *et al.*, "A hybrid graph representation for recursive backtracking algorithms," in *4th International Workshop Frontiers in Algorithmics*, vol. 6213. Springer, 2010, pp. 136–147.

[21] B. Kerbl *et al.*, "The broker queue: A fast, linearizable FIFO queue for fine-granular work distribution on the GPU," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 76–85.

[22] D. S. Johnson *et al.*, *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. American Mathematical Society, 1996, vol. 26.

[23] J. Kunegis, "KONECT: the Koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 1343–1350.

[24] J. Leskovec *et al.*, "SNAP datasets: Stanford large network dataset collection," 2014.

[25] M. A. Dzulfikar *et al.*, "The PACE 2019 parameterized algorithms and computational experiments challenge: the fourth iteration," in *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[26] M. R. Garey *et al.*, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[27] R. E. Tarjan *et al.*, "Finding a maximum independent set," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 537–546, 1977.

[28] J. M. Robson, "Algorithms for maximum independent sets," *Journal of Algorithms*, vol. 7, no. 3, pp. 425–440, 1986.

[29] ——, "Finding a maximum independent set in time $O(2^{n/4})$," *Technical Report - 1251-01, LaBRI, Universite Bordeaux I*, 2001.

[30] M. Xiao *et al.*, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[31] ——, "A refined algorithm for maximum independent set in degree-4 graphs," *Journal of Combinatorial Optimization*, vol. 34, no. 3, pp. 830–873, 2017.

[32] R. G. Downey *et al.*, *Parameterized complexity*. New York: Springer-Verlag, 1997.

[33] J. Chen *et al.*, "Vertex cover: Further observations and further improvements," in *25th International Workshop on Graph-Theoretic Concepts in Computer Science*, P. Widmayer *et al.*, Eds., vol. 1665. Springer, 1999, pp. 313–324.

[34] ——, "Improvement on vertex cover for low-degree graphs," *Networks*, vol. 35, no. 4, pp. 253–259, 2000.

[35] M. R. Fellows *et al.*, "Fixed-parameter complexity and cryptography," in *10th International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, vol. 673. Springer, 1993, pp. 121–131.

[36] J. F. Buss *et al.*, "Nondeterminism within P," *SIAM Journal on Computing*, vol. 22, no. 3, pp. 560–572, 1993.

[37] D. Hespe *et al.*, "WeGotYouCovered: The winning solver from the PACE 2019 challenge, vertex cover track," in *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2020.

[38] T. Akiba *et al.*, "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover," *Theoretical Computer Science*, vol. 609, pp. 211–225, 2016.

[39] M. A. Dzulfikar *et al.*, "The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper)," in *14th International Symposium on Parameterized and Exact Computation*, ser. LIPIcs, vol. 148. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 25:1–25:23.

[40] F. N. Abu-Khzam *et al.*, "Scalable parallel algorithms for FPT problems," *Algorithmica*, vol. 45, no. 3, pp. 269–284, 2006.

[41] ——, "On scalable parallel recursive backtracking," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 65–75, 2015.

[42] K. Toume *et al.*, "A GPU algorithm for minimum vertex cover problems," in *AIP Conference Proceedings*, vol. 1618, no. 1. American Institute of Physics, 2014, pp. 724–727.

[43] M. Burtscher *et al.*, "A high-quality and fast maximal independent set implementation for GPUs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 2, pp. 1–27, 2018.

[44] T. Imanaga *et al.*, "Efficient GPU implementation for solving the maximum independent set problem," in *2020 Eighth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2020, pp. 29–38.

[45] Y. Liu *et al.*, "Finding vertex cover: Acceleration via CUDA," *GPU Technology Conference*, 2008.

[46] X. Chen *et al.*, "Pangolin: An efficient and flexible graph mining system on CPU and GPU," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1190–1205, 2020.

[47] ——, "Sandslash: a two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 378–391.

[48] Y.-W. Wei *et al.*, "Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2352–2366, 2021.

[49] T. Zhang *et al.*, "Optimization of N-Queens solvers on graphics processors," in *International Workshop on Advanced Parallel Processing Technologies*. Springer, 2011, pp. 142–156.

[50] K. Rocki *et al.*, "Parallel minimax tree searching on GPU," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2009, pp. 449–456.

[51] M. Plauth *et al.*, "Using dynamic parallelism for fine-grained, irregular workloads: a case study of the N-Queens problem," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 404–407.

[52] I. El Hajj *et al.*, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[53] M. G. Olabi *et al.*, "A compiler framework for optimizing dynamic parallelism on GPUs," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022.