Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure

Alok Tripathy, Fred Hohman, Duen Horng Chau, and Oded Green

Georgia Institute of Technology

***Abstract*—The k-core of a graph is a metric used in a wide range of applications, including social networks analytics, visualization, and graph coloring. We present two new parallel and scalable algorithms for finding the maximal k-core in a graph. Unlike past approaches, our new algorithms do not rebuild the graph in every iteration – rather, they use a dynamic graph data structure and avoid one of the largest performance penalties of k-core – pruning vertices and edges. We also show how to extend our algorithmsto support k-core edge decomposition for different size k-cores found in the graph. While our new algorithms are architecture independent, our implementations target NVIDIA GPUs. When comparing our algorithms against several highly optimized algorithms, including the sequential igraph implementation and the multi-thread ParK implementation, our new algorithms are significantly faster. For finding the maximal k-core in the graph, our new algorithm can be up-to $58\times$ faster the igraph and up-to $4\times$ faster than ParK executed on a 36 core (72 thread) system. For the k-core decomposition algorithm, we saw even greater and more consistent speedups for our algorithm where it was up-to $130\times$ faster than igraph and up-to $8\times$ faster than ParK. Our algorithms were executed on an NVIDIA P100 GPU.**

I. Introduction

Network graphs are now a ubiquitous data type and model many natural and synthetic phenomena in our modern world. However, analyzing graph data to gain insight into a network remains challenging. In a recent online survey conducted to gather information about how graphs are used in practice, researchers discovered that graph analysts rated scalability and visualization as the most pressing issues to address [1]. Modern day graphs can easily grow to billions of vertices and edges; therefore, as graphs grow in size and become more complex, the need for scalable sense-making algorithms becomes critical for gaining insight into modern day large graphs.

Modern day graph algorithms, for example *edge decomposition algorithms* based on fixed points of degree peeling, show strong potential in helping people explore unfamiliar graph data [2]. This decomposition, based on the well-studied k-core decomposition, has been shown to be useful for graph exploration, navigation, and visualization [3]. The heart of this edge decomposition algorithm requires computing the maximal k-core for a graph. From graph theory, the k-core of a graph is a maximal subgraph in which all vertices have degree at least k. k-core is not only vital to edge decomposition algorithms, but also powers a diverse set of graph exploration tools and systems with applications in large-scale visualization [4], [5], graph clustering [6], hierarchical structure analysis [5], and graph mining [7]. More applications are discussed in II. It has been shown that k-core can be computed in linear time by iteratively removing minimum degree vertices from a graph using a list of vertices per degree [8]. This process of removing minimum degree vertices is commonly called *pruning*, and it is the primary computation by which k-core and edge decompositions rely on.

In this paper, we present two fast and scalable algorithms for finding the maximal k-core of a graph, and extend these to two edge decomposition algorithms for breaking down a graph into smaller subgraphs based on the k-core sizes. Our new algorithms do not require rebuilding the graph after pruning in each iteration of edge composition. Rather, We use a dynamic graph data structure to avoid one of the largest performance penalties of k-core decomposition.

While our new algorithms are architecture independent, our implementations target NVIDIA GPUs. Furthermore, we run extensive experiments on a wide range of graphs, with different topological properties and scales, to evaluate our algorithms. We compare against the current state-of-the-art results found in literature, including several highly optimized algorithms: a sequential igraph implementation and a multi-thread ParK implementation [9].

Contributions

In summary, the contributions of this paper are as follows:

- **Scalable, maximal k-core algorithms.** We introduce two fast and scalable algorithms for finding the maximal k-core of a graph. Both use a dynamic grpah data structure to avoid the penalty of rebuilding the graph after each pruning phase of the algorithm. The first has parallel bottlenecks, but would likely perform well on a sequential processor. The latter performs much better in parallel and on a GPU. When compared with a sequential igraph implementation and a multi-threaded ParK[9] implementation with 72 threads, our second algorithm can be up to $58\times$ faster than igraph and up to $4\times$ faster than ParK (though it is sometimes slower than ParK).
- **Scalable k-core decomposition.** We introduce two different k-core decomposition algorithms for breaking down the graph into smaller subgraphs based on the k-core sizes. These algorithms also use a dynamic graph data structure. Our first algorithm uses a large number of small edge udpates, whereas our second algorithm uses a small number of large edge updates. As a GPU supports thousands of lightweight

threads, our second algorithm performs better in our GPU-based experiments. Specifically, it is up to $130\times$ faster than igraph and up to $8\times$ faster than ParK.

II. Background

A. k-core Applications and Computation

k-core was first introduced by studying social networks [10], but has since seen great attention in a diverse set of other domains. Applications for k-core for graph data include large-scale visualization [4], [5], graph clustering [6], hierarchical structure analysis [5], and graph mining [7]. Other applications that use k-core for understanding particular domains whose data is represented as network graphs include bioinformatics [11], [12], [13], identifying and understanding Internet structure [14], studying the spreading of economic crises [15], identifying influential spreaders in a complex network [16], and recently, revealing hierarchical cortical organization of the human brain [17].

From graph theory, the k-core of a graph is a maximal subgraph in which all vertices have degree at least k. In recent work using k-core, igraph, a network analysis library, is used to perform edge decompsition[3]. However, the igraph implementation of k-core is still sequential [18]. There is, though, a recent multi-threaded implementation of k-core, called ParK, which improves upon the state-of-the-art [9] by significantly reducing the working set size and minimizing the random accesses to the data structure. In our work, we compare our algorithms against both the igraph implementation and ParK and show that we outperform both these algorithms. There is a GPU implementation of k-core vertex decomposition [19] we could not compare as it is not open-source.

B. Edge Decompositions

From [2], edge decompositions based on fixed points of degree peeling divide large graphs into an ordered set of subgraphs that is dependent only upon the topology of the graph. In general, the edge decomposition is computed by finding the maximal k-core of a graph, removing the recently found k-core from the original graph, and repeating until the original graph is empty. Therefore, whereas computing a maximal k-core of a graph relies on graph pruning, computing an edge decomposition relies on multiple maximal k-core computations. Each maximal k-core computed is a fixed point of degree peeling of the graph, i.e., if one were to re-run the edge decomposition on a particular maximal k-core, the decomposition would simply return the original graph, therefore each maximal k-core found in an edge decomposition is fixed. A result of this is that the edge decomposition is deterministic, a useful property for sense-making graph algorithms. In the existing literature for applying edge decompositions to large scale graph visualization and exploration, each maximal k-core is called a *graph layer* [4], [3]. Graph layers help users identify potentially important substructures (e.g., quasi-cliques, multi-partite-cores), by automatically separating such patterns from the majority of the graph. Our algorithms and implementations presented in this paper will allow future researchers to decompose large graphs faster to better understand networks and gain insights into their complex internal structure.

TABLE I: *List of symbols and notations used by our algorithm*

Symbols and Notations	
Symbol	Description
G	Input graph.
$V(G)$	Set of vertices in the input graph.
$E(G)$	Set of edges in the input graph.
$\widehat{G}$	Separate graph, used for storage in HKS/HDS.
Q	Queue of vertices removed per $peel$.
V_b	Batch of vertices to delete.
E_b	Batch of edges to delete.
K	K-core of G
Notations and Fields	
$color[v]$	True if v will be pruned this iteration in HKS/HDS, false otherwise.
$flag[v]$	True if v does not exist in G in HKO/HDO, false otherwise.
$peel$	Max degree value used to determine whether to prune a vertex.

C. Dynamic Graph Data Structures

Dynamic graph data structures, as opposed to static graph data structures, deal with graphs that change. Dynamic can imply that these changes are temporal. Yet, in fact changes can also be structural, meaning that the structure of the graph changes. This is the case with k-core where vertices and edges are pruned from the graph (repeatedly). This process can also be found for other problems such k-trusses and [20]. Dynamic graph data structures, described below, can help avoid recreating the graph in every step of the computation as was done in [2], [4], [3]. Using a dynamic graph data structure avoids these overheads.

The Hornet [21] data structure is a dynamic graph data structure designed for dealing with fast and parallel updates to the graph. Specifically, Hornet was designed to process numerous insertion and deletion of a large number of vertices and edges. These types of operations are especially important for practical purposes as an inefficient dynamic graph data structure will greatly reduce the overall performance of the graph algorithm. Hornet support over 150 million updates per second on current GPU systems. While our implementation uses the Hornet data structure, we note that our algorithm can be implemented for other dynamic graph data structures so long as they support vertex and edge insertions and deletions. The reader is referred to [21] for more details on the Hornet data structure and a wider literature survey of dynamic graph data structures.

a) Batches: Specifically, batches refer to the fact that there multiple updates are made to the graph and the order in which they are processed is not important. In the case of k-core we will show that all the vertices and edges pruned in every iteration can actually be placed into a single batch and deleted concurrently.

III. k-core Number Algorithms

In this section we present our new algorithms for finding the largest $k-core$ in a graph. While there has been extensive research in designing algorithms for finding the k-core numbers, our algorithms are unique as these are the first, to the best of our knowledge, to take advantage of a dynamic graph data structure. Whereas many past algorithms need to rebuild the graph for each iteration of the k-core search, our algorithm utilizes a data structure designed for dynamic graphs that is highly optimized for edge insertions and deletions. This

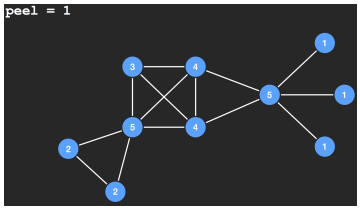


FIG. 1: *Finding* $deg = peel = 1$ *vertices*

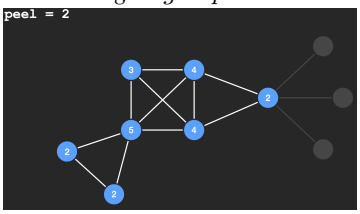


FIG. 2: *Incrementing* $peel = 1$ *when no* $deg = 1$ *vertices remain*

Algorithm 1 K-Core Slow - first algorithm for finding the maximal k-core.

```
1: peel ← 1
2: Q ← {}
3: Ĝ ← ({}, {})
4: while |V(G)| > 0 do
5:     color[v] ← 0∀v ∈ V(G)
6:     V_b ← {}
7:     parallel for v ∈ V(G) do
8:         if deg[v] ≤ peel then
9:             color[v] ← 1
10:            V_b.enqueue(v)
11:    end parallel for
12:
13:    if |V_b| > 0 then
14:        E_b ← {}
15:        parallel for (u, v) : u ∈ V_b, v ∈ adj(u) do
16:            if color[u] ∧ color[v] then
17:                E_b.enqueue((u, v))
18:        end parallel for
19:        G.delete_edges(E_b)
20:        G.delete_vertices(V_b)
21:        Ĝ.insert_vertices(V_b)
22:        Ĝ.insert_edges(E_b)
23:        Q ← Q ∪ V_b
24:    else
25:        peel ← peel + 1
26:        Q ← {}
27: return (induced_subgraph(Ĝ, Q), peel)
```

allows us to avoid the overhead of rebuilidng the graph in each iteration of the algorithm. As we will show in this section, in each iteration of k-core, edges that no longer meet the k-core requirements are pruned from the graph using these dynamic graph operations.

In the next section (Sec.IV) we show that these operations are in fact even more important as the k-core algorithm will be executed a many times. Here we will show two different approaches for finding the k-core number of a static graph using dynamic graph operations. Each approach offers a different set of advantages and disadvantages, primarily regarding parallelization. Our first algorithm opts for a large number of small edge batches, and small edge batches need fewer cores for parallelization. There is also a fair amount of synchronization in our first algorithm.

Our second algorithm makes a small number of large edge batches. These large batches are more easily parallelizable. Furthermore, our second algorithm does not need as much synchronization, making it better-suited for GPU acceleration.

Algorithm 2 K-Core Optimized - second algorithm for finding the maximal k-core.

```
1: peel ← 1
2: Q ← {}
3: num_active = |V(G)|
4: color[v] ← 0∀v ∈ V(G)
5: deg[v] ← G.deg(v)∀v ∈ V(G)
6: while num_active > 0 do
7:     V_b ← {}
8:     parallel for v ∈ V(G)∧!flag[v] do
9:         if deg[v] ≤ peel then
10:            flag[v] ← 1
11:            V_b.enqueue(v)
12:    end parallel for
13:    Q ← Q ∪ V_b
14:    num_active ← num_active − |V_b|
15:
16:    if |V_b| > 0 then
17:        parallel for (u, v) : u ∈ V_b, v ∈ adj(u) do
18:            deg[u] ← deg[u] − 1
19:            deg[v] ← deg[v] − 1
20:        end parallel for
21:    else
22:        peel ← peel + 1
23:        Q ← {}
24: return (induced_subgraph(G, Q), peel)
```

A. First Algorithm

At a high level, our first algorithm works as follows. It starts off by finding the k-core number of G by incrementally removing vertices and their incident edges that do meet the core requirement. Specifically, the algorithms starts with $k = 1$. This value is incremented as described below. When a vertex is marked for deletion from G, its respective incident edges will also be removed. We begin by repeatedly removing all vertices with degree less than 1 and their edges until there are no longer degree less than 1 vertices in G as part of a *batch*. Once there are no longer vertices with degree 1 in G, we begin removing vertices with degree at most 2. This is illustrated in Figure 1.

The removal process is continued until there are no more vertices and edges in the graph. While it might seem that the graph is empty and as such we have lost the largest k-core, we in fact have the set of vertices and edges removed in the last (and previous) iteration. That is, if we reach an empty graph with $peel = k$, then all vertices removed when searching for vertices with degree at most k form the largest k-core of G. To find these vertices, whenever we remove vertices with degree $peel$, we insert these vertices into a queue Q as shown on line 23 in Algorithm 1. This queue is emptied when we begin searching for vertices with degree $peel + 1$ on line 25. Thus, once there are no vertices in G remaining, the vertices in the queue form the largest k-core of G, and the k-core number of G is $peel$.

Note that while the queue has the vertices in the largest k-core of G, we do not have the edges of the k-core. The edges of the k-core are the edges in the induced subgraph formed by the vertices of the k-core. However, G is now empty, so it is impossible to determine what those edges are. To solve this, we maintain an additional graph $\widehat{G}$. $\widehat{G}$ begins empty, and whenever a vertex and its edges are deleted from G, we insert them into $\widehat{G}$, as depicted on lines 21 and 22. This way, once each vertex and edge is removed from G, $\widehat{G}$ will be equal to what G was at the beginning of the algorithm. We can find the induced subgraph in $\widehat{G}$ to find the largest k-core.

There are certain implementation details that are critical so

this can be parallelized well. For instance, when we iterate over all vertices to look for those of degree $peel = k$, we do not remove vertices immediately. Rather, we simply color any vertex with degree at most k for now. Then, in a separate loop, we iterate over all edges in G. Any edge with at least one colored endpoint is added to a batch to be deleted.

B. Second Algorithm

Recall that this algorithm should perform much more efficiently on a GPU than the former. It has larger parallel sections with cheap computation, and requires less synchronization.

Similar to the previous algorithm, we start off this algorithm in the same way as above, initializing $peel = 1$ and finding all vertices with degree $peel = 1$.

Earlier we would find all vertices with degree $peel = 1$, and delete them with their incident edges from G. Now, however, we do not delete the vertices and edges. Instead, when we find a vertex u, we flag u and decrement the degree of u and all of u's neighbors. Vertices that are flagged are not considered to be present in G, although they are not explicitly deleted. This is depicted in Algorithm 2.

The remainder of this algorithm is similar to the first. We terminate once all vertices in G have been flagged, analogous to terminating when all vertices in G have been deleted. Once the algorithm terminates, the vertices flagged with the most recent $peel$ form the largest k-core in G. To keep track of those vertices, we maintain a queue of removed vertices that empties each time $peel$ is incremented. The k-core in G will then be the subgraph of G induced by the vertices in the queue, and the k-core number is $peel$. Note that we do not need $\widehat{G}$ as we do not remove any vertices or edges from G.

C. Complexity Analysis

1) Work Complexity: We note the following for both algorithms, every edge is accessed exactly only once. The only time an edges is accessed is prior to its removal. Each edge is in fact accessed only after the source vertex of the edge has been marked for removal; this can help no more than once. Thus, the work complexity for both is $O(cV + E)$, where c is the number of iterations in the algorithm. In most cases c is small. Note that in the worst case, we have $O(V)$ iterations, since each iteration enqueues at least one vertex. The worst-case work complexity for both algorithms is, thus, $O(V^2+E)$.

2) Storage Complexity: The first algorithm has a few $O(V)$ arrays and $O(E)$ arrays. $\widehat{G}$ will also contribute space. In theory it should not as every edge inserted into $\widehat{G}$ has been deleted from G, but in practice extra space is used to prevent excessive reallocating. This is an extra $O(E)$ space in worst-case. Thus, the overall storage complexity adds up to $O(V + E)$ in worst-case, although this worst-case bound requires a large E_b batch. Our second algorithm only uses arrays of length $O(V)$, so it has $O(V)$ storage complexity.

IV. k-core Decomposition Algorithms

Every edge in the graph can belong to several k-cores (of different sizes). The k-core edge decomposition of a graph will find the largest k-core that each edge belongs to. The decomposition for a specific value of k is also known as the peel.

Algorithm 3 K-Core Decomposition Slow - first algorithm for finding the k-core decomposition algorithm.

```
1: Ĝ ← ({}, {})
2: while |V(G)| > 0 do
3:     K, k_num ← KcoreNum1(G, Ĝ)
4:     parallel for e ∈ E(K) do
5:         peels[e] ← k_num
6:     end parallel for
7:     Ĝ.delete_edges(E(K))
8:     Ĝ.delete_vertices(V(K))
9:     swap(G, Ĝ)
10: return peels[]
```

Algorithm 4 K-Core Decomposition Optimized - first algorithm for finding the k-core decomposition algorithm.

```
1: while |V(G)| > 0 do
2:     K, k_num ← KcoreNum2(G)
3:     parallel for e ∈ E(K) do
4:         peels[e] ← k_num
5:     end parallel for
6:     G.delete_edges(E(K))
7:     G.delete_vertices(V(K))
8: return peels[]
```

Computing the k-core decomposition is, however, more computationally demanding that just finding the maximal k-core as it also includes just finding the maximal k-core in addition to the proceeding k-cores. This is especially true for large graphs with millions of vertices and billions of edges. For such graphs the number of iterations to run k-core is equal to the number of graph layers produced by the edge decomposition, which could be in the hundreds [2], [4] The current state of the art algorithms generally do not process large graphs quickly due to several key constraints: 1) they are not taking full advantage of massively multi-threaded systems and 2) the primarily use inefficient data structures for representing the graph. If the data structure is storage efficient, such as CSR, it is typically immutable and will require rebuilding the graph in every iteration. Other mutable data structures such as edges lists are possible but then these lose locality and are not computationally efficient.

A. K-core Decomposition

The following outlines the approach we take for decomposing the graph into various k-cores. First of all, find the maximal k-core of the graphs (in terms of k) using either Algorithm 1 or Algorithm 2. Then, for all edges in the this k-core, set their $peel$ value to k. Then remove the k-core from the graph while storing these edges in a map $peels[]$ with the peel values. These three steps are done in an iterative manner until the graph is empty. These steps are also illustrated in our k-core decomposition algorithms: Algorithm 3 and Algorithm 4.

Our first k-core decomposition algorithm uses the first k-core number algorithm, and our second k-core decomposition algorithm uses the second k-core number algorithm. Numerous differences in our k-core decomposition algorithm are because of the corresponding k-core number algorithms. Our first algorithm has several small vertex and edge batches for our

dynamic graph data structure. When accelerated on a multi-threaded system, this will show poor scalability, although this perform well on a small number of cores (or threads). Our second algorithm, on the other hand, has fewer but larger batches of vertices and edges, and more SIMD parallelization, making it well-suited for GPU acceleration.

Note that our dynamic graph data structure is a critical component for these algorithms. Both algorithms make heavy use of dynamic graph operations. Namely, they use a considerable amount of edge and vertex insertion and deletion. Performing these algorithms on a data structure meant for static graphs would prove to be a bottleneck for the algorithms, since a lot of time will be spent on copying, memory allocation, and sequentially accessing each edge or vertex in the batch. Thus, a dynamic graph data structure not only is the appropriate structure, but also one that is imperative for performance for both algorithms. As the data structure we will use for experimentation is best-suited for the GPU, our second algorithm will show to benefit the most from the data structure since it has more SIMD parallelization.

B. First Algorithm

Our first k-core decomposition algorithm is an adaptation of our first k-core number algorithm. We refer to this algorithm as K-Core Decomposition Slow. We repeatedly call the k-core number algorithm, modify G, call the k-core algorithm on the modified version of G, and repeat. In this algorithm we use two graphs, the first graph is the original input graph and the second graph will be all the vertices and edges removed from the first graph during the pruning process for finding the maximal k-core. Thus, the second graph inserts the edges deleted from the first graph–both of these are dynamic graph operations.

1) Finding the largest k-core - First, we find the largest the largest k-core in G simply by calling our first k-core number algorithm. *2) Setting $peel$ values* - In the next phase, we iterate over the edges found by the maximal k-core algorithm and mark them with the value of that k-core (lines $4-6$ of 3)–**this array stores the output of the algorithm**. *3) Removing k-core from G* - After marking all the edges found in the current maximal k-core, we can remove them from the second graph $\widehat{G}$ (lines $7-8$ of 3). Recall, that in the process of finding the maximal k-core, we have removed all the edges from G and such it is empty. This leads us to our final phase. *4) Reiterating on the remainder of the graph* - Thus, to continue finding k-cores, we need to continue iterating over the remaining edges and we will swap the vertices and edges in G and $\widehat{G}$ [1].

C. Second Algorithm

Our second k-core decomposition algorithm is an adaptation of our HKO, second maximal k-core algorithm. We refer to this decomposition as the optimized algorithm, or K-Core Decomposition Optimized as it removes many unnecessary operations/ Much like the first algorithm, we call a k-core number algorithm, set $peel$ values, remove the k-core, and reiterate until the graph is empty. Yet, the individual step are slightly different as we only require a single graph (rather than the two graphs needed by K-Core Decomposition Slow).

1) Finding the largest k-core - We find the largest the largest k-core in G simply by calling our HKO k-core number algorithm (line 2 of 4). *2) Setting $peel$ values* - Our second k-core number algorithm also returns both the k-core number and the corresponding k-core. Thus, we can simply iterate over all edges of the k-core in parallel and assign the k-core number as the $peel$ for each edge. Once again, **this array stores the output of the algorithm**. *3) Removing k-core from G* - Removing the k-core from G is simpler in our second k-core decomposition algorithm as no edges were removed in the maximal k-core functions. This leads to fewer and larger edge removals from our dynamic data structure. *4) Reiterating on the remainder of the graph* - After removing the k-core from G, we can simply reiterate on G. This step is also simpler for our second k-core decomposition algorithm.

D. Comparison

1) Complexity Analysis: First, note that both our new decomposition algorithms give the same output and have the same number of iterations. The number of iterations is dependent on the graph topology.

The complexity of a single iteration of each k-core decomposition algorithm is dominated by the finding the k-core number algorithm, as there is not much work done outside of calls to k-core number functions. Thus, both k-core decomposition algorithms have a work complexity of $O(V^2 + E)$ for one iteration.

2) Algorithm Scalability: Both algorithms have a good amount of parallelism available. Specifically, the second k-core decomposition algorithm tends to scale better as it uses a smaller number of large batches used for the dynamic graph data structure. In contrast, the first algorithm uses a larger number of small batches. For massively multi-threaded systems, using smaller batches can underutilize the system due to workload imbalance, synchronization, and communications overheads.

E. Sequential Vs. Parallel Algorithms

In this section we presented two algorithms for k-core decomposition. Throughout the discussion, we did not discuss the architecture on which these algorithms are executed as they are in fact architecture agnostic. Specifically, our implementation of these algorithms targets NVIDIA's GPU, a massively multithreaded system that can support thousands of light weight threads. Effective utilization of this system requires good load-balancing and ensuring that this is enough work to be dispatched to these threads. In practice, we found that our first algorithm lacked the scalability needed to utilize thousands of threads in each phase of the algorithm. This led to the development of the second algorithm which in practice significantly outperforms the first. For a sequential implementation it could be that the first algorithm will perform

[1] In practice, this inexpensive and is equivalent to pointer swapping.

TABLE II: *Systems used in experiments.*

Architecture	Micro-architecture	Processor	Frequency	Cores	Threads	LL-Cache / Total Bandwidth	DRAM Size & Type
CPU x86-64	Broadwell	Xeon E5-2695	2.1 GHz	36	72	45 MB LLC	1007GB DDR4
GPU Tesla	Pascal	P100 PCIe	1.13 GHz	56	3584	732GB/s Bandwidth	16GB HBM2

TABLE III: *List of networks used in experiments.*

Name	$\|V\|$	$\|E\|$	k-core number
dblp-author	5,425,964	8,649,002	10
patentcite	3,774,769	16,518,947	64
soc-LiveJournal1	4,847,571	42,851,237	372
soc-pokec-relationships	1,632,804	22,301,964	29
trackers	27,665,731	140,613,747	438
wikipedia-link-de	3,225,566	65,759,634	829

as well as the second. However, we are unable to compare these as we do not have an optimal data structure for the CPU.

V. EXPERIMENTAL SETUP

In the following section we highlight our experimental setup for checking the performance of our new algorithms. This includes the systems, networks, and various benchmarks used.

Our new algorithms were benchmarked on an NVIDIA P100 GPU connected to an Intel Xeon E5-2695 with 36 cores 72 threads (details in Table II). The P100 is a Pascal based GPU with 56 SMs and 64 SPs per SM, for a total of 3584 SPs (lightweight threads). The P100 has a total of 16GB of HBM2 memory. The Intel Xeon E5-2695 is a Broadwell based processor running at 2.1 GHz with 45MB L3 cache. The server consists of two such processors with a total of 1TB of memory. While our new algorithms are architecture independent, the final implementation targets the GPU as the Hornet [21] data structure we used targets the GPU.

A. Dataset

For experiments we use a wide range of graphs taken from the Konect graph database [22], though some of these graphs are originally from SNAP [23]. Details of these graphs can be found in Table III. Note, we also include the size of the maximal k-core for each of these graphs. The benchmarks, detailed below, require the input in a different format (edge list and binary format). We ensure that benchmarks receive the correct inputs and also do not include the time needed to read the file or pre-process it.

B. Benchmarks

We compare the performance of our algorithms with two highly optimized implementations. Specifically, we compare our algorithms against the sequential algorithm found in [24] which is also implemented in the igraph library [18]–we refer to this algorithm simply as *igraph* in our comparison. We also compare against the multi-threaded ParK algorithm [9] which extends the igraph algorithm the multi-threaded system. ParK was able to use all 36 cores and 72 threads of the CPU used in our experiments. In many cases, ParK shows good scalability in comparison to its sequential counterpart, yet our new algorithms are significantly faster.

The igraph library has an implementation of the BZ algorithm as part of its $coreness$ function. However, recall that BZ computes the k-core vertex decomposition of G. That is, it returns an array with the largest k-core each vertex in G is contained in, not each edge. Our k-core number implementation with the igraph library simply calls this $coreness$ function and returns the largest value in the array. Our k-core edge decomposition implementation with igraph uses the above as the k-core number implementation, removes the corresponding k-core from G, and repeats.

TABLE IV: k-*core number times in seconds.*

Name	HKO	HKS	ParK	igraph
dblp-author	0.028 58×	0.731 2.2×	0.105 15.×	1.633 1×
patentcite	0.147 26×	2.953 1.3×	0.253 15×	3.825 1×
soc-LiveJournal1	0.838 7.4×	OOM OOM	0.549 11.3×	6.191 1×
soc-pokec-relationships	0.174 15×	4.331 0.6×	0.155 16.6×	2.586 1×
trackers	13.160 1.6×	OOM OOM	3.052 6.8×	20.693 1×
wikipedia-link-de	1.987 2×	OOM OOM	0.764 5.1	3.954 1×

C. New Algorithms

Recall, for both finding the maximal k-core algorithm and for the k-core decomposition we presented two algorithms. In both cases, the second algorithm was developed due some of the performance flaws found in the first algorithm when on the GPU. We show results for both the slower and faster instances and compare their performance behavior in addition to the external benchmarks we used. We note that the slower of our implementations might in fact be suitable for a sequential or multi-threaded environment when the number of executing threads is not to high, but not for systems with high thread counts.

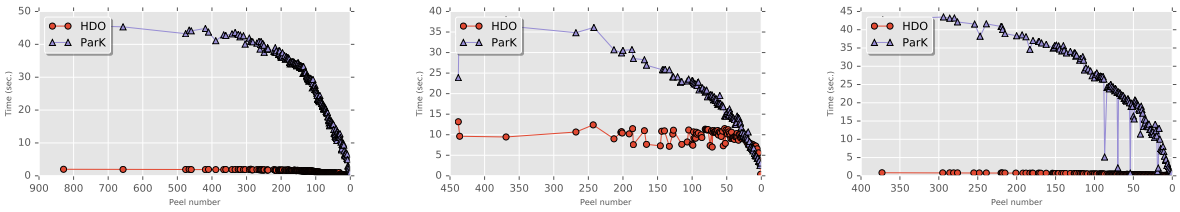
For finding the maximal k-core we denote our two algorithms, Algorithm 1 and Algorithm 2 as HKS and HKO, respectively. HKS refers to Hornet K-Core Slow and HKO refers Hornet K-Core Optimized. For the k-core decomposition we denote our two algorithms, Algorithm 3 and Algorithm 4 as HDS and HDO, respectively. HDS refers to Hornet Decomposition Slow and HKO refers Hornet Decomposition Optimized.

D. Hornet

Our implementation uses the highly-optimized Hornet data structure [21]. Hornet is the fastest dynamic graph data structure available for shared-memory systems and can handle over 100 million edges per second for large updates.

VI. PERFORMANCE ANALYSIS

a) Execution Time Analysis: Table IV and Table V depict the execution for finding the maximal k-core of the graph and for k-core decomposition, respectively. In both tables we compare our new algorithms against the multi-threaded ParK algorithm and the sequential igraph algorithm. Note, all these timings ignore the time it takes to read the input from disk and the time it takes to initialize the graphs. Instead we focus



(a) $wikipedia - de$ (b) $trackers$ (c) $soc - LiveJournal1$

FIG. 3: *Time per peel for k-core decomposition. Notice that the peels are placed in reversed order as the decomposition starts with the largest peels first. Lower is better.*

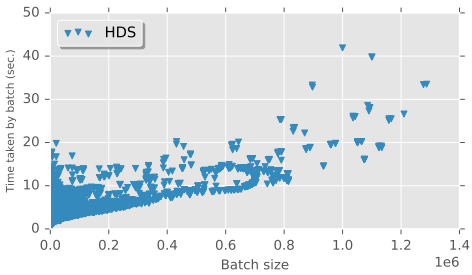
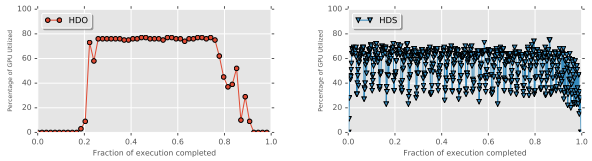


FIG. 4: *Batch Size vs. Time (ms.) on $soc - pokec - relationships$. This plot highlights that smaller batch updates are slow and are the performance bottlenecks for HDS as they under-utilize the GPU. Its through this analysis that we designed HKO and HDO to overcome the small batch problem.*



(a) *HDO* (b) *HDS*

FIG. 5: *GPU Utilization vs. Time (sec.) for $patentcite$ normalized across execution.*

on the time it takes to do the k-core algorithms. In [21], it was shown the creating a Hornet data structure is not much more expensive than creating a CSR data structure. Also, the time it takes to build the graphs is fairly small in comparison to the time spent finding the maximal k-core or decomposing the graph. We discuss in more detail in below (and show experimental results in Fig. 5). Lastly, Figure 3 depicts the execution time taken per iteration for both HDO and ParK for the k-core decomposition. Lower times are better. For the higher peels (beginning of the decomposition) our algorithm can be more than $10\times$ faster than ParK. In the later iterations, our algorithm is still faster though the speedup is not as high as there is less work and we are unable to utilize the GPU to its full capabilities.

We start off by analyzing the slower of our algorithms, HDS and HKS. HDS is faster than the igraph by about $3.5\times - 13\times$,

TABLE V: *k-core decomposition times in seconds.*

Name	HDO	HDS	ParK	igraph
dblp-author	0.635 129.2×	6.184 13.3×	1.595 51.5×	82.066 1×
patentcite	5.200 63.8×	91.481 3.6×	13.294 25×	331.538 1×
soc-LiveJournal1	60.755 25.9×	OOM OOM	487.112 3.3×	1572.985 1×
soc-pokec-relationships	2.756 85.6×	50.049 4.7×	6.488 36.3×	235.790 1×
trackers	1006.954 4.692×	OOM OOM	1148.638 4.113×	4725.317 1×
wikipedia-link-de	266.923 11.3×	OOM OOM	1397.323 2.149	3003.166 1×

but is $2\times - 4\times$ worse than ParK. HDS is faster than igraph due its parallel scalability and its ability to utilize the GPU. HKS is occasionally faster than igraph, and if so only be $1.1\times - 2\times$, and is worth than ParK by $6\times - 32\times$. The key bottleneck in HKS is the large number of updates made to Hornet, the dynamic graph data structure. This also becomes a bottleneck in HDS, as HDS calls HKS numerous times. Small batches underutilize the GPU and add considerable overheads such as kernel launch overhead. This leads to the slowdowns in comparison to ParK. To overcome these slowdowns algorithms HDO and HKO were designed to avoid these performance penalties.

HDO performs quite well compared to the CPU algorithms, $2\times - 8\times$ faster than ParK (on 36 cores / 72 threads) and $11\times - 130\times$ faster than the sequential and optimized igraph algorithm. HKO also performs well, hovering between $2\times - 58\times$ faster than igraph, and is sometimes faster than ParK by $2\times - 4\times$. Unlike HDS, HDO is able to fully utilize the GPU. We discuss the GPU utilization below.

Note that while HKO occasionally beats ParK, HDO is consistently faster than both ParK and igraph by a fair amount. The reason HDO generally outperforms ParK, despite having a higher work complexity, is the massive amount of parallelization available and the dynamic graph data structure. The benefits of the dynamic data structure are only available to HDO which actually does the edge deletions–HKO does not remove any edges. Without this dynamic graph data structure, deletions end up being prohibitive and require rebuilding the graph which is what ParK is doing. This leads to HDO our performing better than ParK for k-core decomposition.

b) HDS and HKS analysis: To verify that the bottleneck for HDS and HKS is numerous small batches, we observe the time taken by the batches within HKS as a function of batch size in Figure 4. Note that this is not linear. Most of

the batches fall under 200000 edges, and that many of these batches take roughly 10ms, no less time than some batches of size $600000-800000$. This implies that the execution of these batches is dominated by overheads such as kernel instantiation. This is investigated further in Para. VI-0c.

c) GPU Utilization: We also plot GPU utilization for the $patentcite$ network for both HDS and HDO, Fig. 5. To measure the times, we run the *nvidia-smi* tool concurrently to our algorithm and sample the utilization. As this adds some overhead to the execution time, we normalize [2] the execution time between 0 and 1. For both HDS and HDO, these plots also includes the time it takes to load the graph from disk, create the graph in memory, and write the results to disk. For this reason, its possible to the GPU at 0% utillization at the beginning and end of the execution. Note, due to normalization its not possible to see that the HDO's execution time is roughly $12\times$ lower than the that of HDS.

For the HDO algorithm, the GPU utilization is consistently above 80% and is many cases is above 90%. In the later phases of HDO, the utilization goes down as the graph has been greatly reduced and few vertices and edges remain. This is not the case for the slower HDS algorithm which suffers from lower utilization due to a larger number of small updates made to the graph. The sample points for HDS hover in the range of $20\% - 75\%$ utilization.

VII. Conclusions

In this paper we presented several new algorithms for finding both the maximal k-core of a graph as well as decomposing the graph into smaller subgraphs based on various k-core values. Finding the maximal k-core and decomposing the graphs into smaller subgraphs are ubiqitious and used across a wide range of problems: structural analysis, visualization, and graph clustering. For finding the maximal k-core we showed that our new algorithm can be up-to $4\times$ faster than ParK and $58\times$ faster than igraph. For the k-core decomposition our new algorithm was upto $8\times$ faster than ParK and $130\times$ faster than igraph. As graphs continue to grow in size faster and more scalable solutions become necessary. Our new algorithms meet these requirements and can be executed on a massively multi-threaded systems. We show a detailed performance analysis on an NVIDIA GPU and show that our algorithm can be executed across thousands of threads.

References

[1] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, 2017.

[2] J. Abello and F. Queyroi, "Network decomposition into fixed points of degree peeling," *Social Network Analysis and Mining*, vol. 4, no. 1, pp. 1–14, 2014.

[3] J. Abello, F. Hohman, and D. H. Chau, "3d exploration of graph layers via vertex cloning," in *2017 IEEE Conference on Visual Analytics Science and Technology (VAST), Poster*. IEEE, 2015.

[2] We ran the same experiment multiple times with and without *nvidia-smi* to verify that this was indeed overhead added due to concurrent executions of programs on the GPU and saw nearly identical performance for those executions

[4] J. Abello, F. Hohman, V. Bezzam, and D. H. Chau, "Large graph exploration via subgraph discovery and decomposition," *arXiv preprint arXiv:1808.04414*, 2018.

[5] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases," *arXiv preprint cs/0511007*, 2005.

[6] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis, "Corecluster: A degeneracy based graph clustering framework." vol. 14, 2014, pp. 44–50.

[7] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "Corescope: Graph mining using k-core analysispatterns, anomalies and algorithms," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 469–478.

[8] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[9] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 9–16.

[10] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[11] G. D. Bader and C. W. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC bioinformatics*, vol. 4, no. 1, p. 2, 2003.

[12] M. Altaf-Ul-Amine, K. Nishikata, T. Korna, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori *et al.*, "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics*, vol. 14, pp. 498–499, 2003.

[13] S. Wuchty and E. Almaas, "Peeling the yeast protein network," *Proteomics*, vol. 5, no. 2, pp. 444–449, 2005.

[14] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A model of internet topology using k-shell decomposition," *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, pp. 11 150–11 154, 2007.

[15] A. Garas, P. Argyrakis, C. Rozenblat, M. Tomassini, and S. Havlin, "Worldwide spreading of economic crisis," *New journal of Physics*, vol. 12, no. 11, p. 113043, 2010.

[16] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, p. 888, 2010.

[17] N. Lahav, B. Ksherim, E. Ben-Simon, A. Maron-Katz, R. Cohen, and S. Havlin, "K-shell decomposition reveals hierarchical cortical organization of the human brain," *New Journal of Physics*, vol. 18, no. 8, p. 083013, 2016.

[18] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[19] H. Zhang, H. Hou, L. Zhang, H. Zhang, and Y. Wu, "Accelerating core decomposition in large temporal networks using gpus," in *Neural Information Processing*, 2017, pp. 893–903.

[20] J. Cohen, "Graph Twiddling in a Map-Reduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[21] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[22] J. Kunegis, "Konect – the koblenz network collection." *Proc. Int. Conf. on World Wide Web Companion*, pp. 1343–1350, 2013.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[24] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003. [Online]. Available: http://arxiv.org/abs/cs.DS/0310049