2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00142

Accelerating k-Core Decomposition by a GPU

Akhlaque Ahmad†, Lyuheng Yuan*, Da Yan*, Guimu Guo‡, Jieyang Chen*, Chengcui Zhang*

*University of Alabama at Birmingham, Birmingham, AL, USA {lyuan, yanda, jchen3, czhang02}@uab.edu
†TED University, Ankara, Turkey akhlaque.ahmad@tedu.edu.tr ‡Rowan University, NJ, USA guog@rowan.edu

***Abstract*—The k-core of a graph is the largest induced subgraph with minimum degree k. The problem of k-core decomposition finds the k-cores of a graph for all valid values of k, and it has many applications such as network analysis, computational biology and graph visualization. Currently, there are two types of parallel algorithms for k-core decomposition: (1) degree-based vertex peeling, and (2) iterative h-index refinement. There is, however, few studies on accelerating k-core decomposition using GPU. In this paper, we propose a highly optimized peeling algorithm on a GPU, and compare it with possible implementations on top of think-like-a-vertex graph-parallel GPU systems as well as existing serial and parallel k-core decomposition algorithms on CPUs. Extensive experiments show that our GPU algorithm is the overall winner in both time and space. Our source code is released at https://github.com/akhlaqueak/KCoreGPU.**

***Index Terms*—GPU, k-core, graph, h-index**

I. Introduction

Networks are ubiquitously used to model interacting entities in modern applications, such as social networks, biological networks, and knowledge graphs. These networks are often huge, so it is important to accelerate their analysis using modern hardware such as GPU with thousands of cores.

One popular tool for network analysis is k-core decomposition [70]. Formally, the k-core of a graph $G = (V, E)$ is the largest induced subgraph with minimum degree k (i.e., where every vertex has degree $\geq k$). For example, Fig. 1 shows the 1-core, 2-core and 3-core of a graph. Specifically, the 2-core contains all the yellow and red nodes in the yellow dashed contour, since it is the largest induced subgraph where every vertex has degree ≥ 2, as any green vertex has degree 1. Note that even though vertex A has degree 3, it is not in 3-core since its neighbor B has degree 2 so cannot be in 3-core, hence A has at most 2 neighbors in 3-core. k-core decomposition finds the core number of every $v \in V$, denoted by $core(v)$, which is the largest value of k that v belongs to a k-core. For example, $core(A) = 2$ in Fig. 1 since A is in 2-core but not 3-core.

Applications of k-core decomposition include detecting dense social communities [65], [66], finding influential spreaders [55], detecting protein interactions [28], analyzing gene networks [36], and understanding the Internet topology [27], [34]. Moreover, since k-core decomposition can be computed in linear time [30], it often serves as an effective lightweight preprocessing to prune unpromising vertices when computing denser structures whose computations have a much higher time complexity [39], [43], [48], [49], [52], [67], [83].

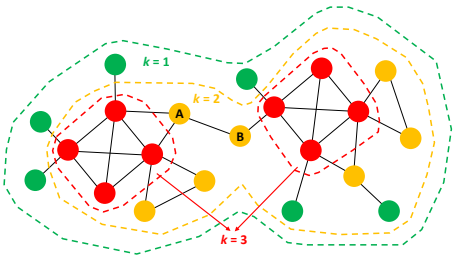

Fig. 1. Illustration of k-Cores

Intuitively, k-core decomposition can be computed in linear time [30] using the serial peeling algorithm of Batagelj and Zaversnik [30], called **BZ** hereafter, which removes the vertex with the smallest degree from G each time. For example, Fig. 1 gives the 1-core, and as we remove all degree-1 vertices that are green, we obtain 2-core; and as we remove all degree-2 vertices one at a time (e.g., B will be removed so that A has degree 2 and then removed), all yellow vertices will be removed and we obtain 3-core. The parallelization of BZ is, however, not straightforward. ParK [41] is a pioneering parallel algorithm that adapts the peeling algorithm to run in a shared-memory multicore environment, and it is later improved by PKC [51] to reduce synchronization overhead.

Montresor, De Pellegrini, and Miorandi [63] propose a very different distributed algorithm, called **MPM** hereafter, where every vertex repeatedly performs h-index-style local updates to estimate its core number from the latest core-number estimates of its neighbors until convergence. In MPM, each vertex may compute for multiple times, so the total workload is higher than BZ and its parallel counterparts such as PKC where each vertex computes only once. MPM was found to be slower than PKC when implemented in a shared-memory multicore environment [51]. However, MPM has been favored and popularly adopted in the latest research [56], [79] for distributed computation, since MPM allows all vertices to conduct local updates simultaneously with minimal dependency.

Despite the increasing availability of GPUs, there are few GPU-based tools for k-core decomposition. Specifically, VETGA [60] is currently the fastest GPU implementation of k-core decomposition. It follows the peeling paradigm, but reframes the problem in terms of vector primitives so that it can be executed using highly optimized GPU vector processing operations in PyTorch. This approach was found to be orders of magnitude faster than the other GPU-based algorithm [75] we are aware of, which uses a suboptimal approach that peels from the highest core number to the lowest, in contrast to BZ that peels from the lowest core number. The implementation of [75] is also not publicly released for use by graph analysts.

In this paper, we study the implementation of both BZ- and MPM-style parallel algorithms (that have been proven effective in a CPU environment) for execution directly on a GPU. While PKC has proven to be faster than MPM-style algorithm in a

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00142

multicore machine [51], it is still interesting to compare them in the GPU setting since a GPU has thousands of threads and thus supports massive parallelism, which could offset the higher total workload of MPM given its minimal dependency (hence high concurrency) for vertex local updates.

Unlike [60] that relies on PyTorch for GPU execution, we develop a tailor-made PKC counterpart directly using CUDA to achieve native performance. Moreover, we notice that both the peeling algorithm and the h-index-style MPM algorithm can be formulated using a think-like-a-vertex paradigm, and can thus be implemented with vertex-centric GPU systems for graph processing, such as Medusa [85], Gunrock [77] and GSWITCH [61]. We, therefore, provide implementations of all the above discussed methods and compare them via extensive empirical studies to provide insights. All our implementations are released at https://github.com/akhlaqueak/KCoreGPU.

Our main contributions are summarized as follows:

- We develop a peeling algorithm on GPU to achieve the native performance, and we implement many optimization techniques on top to verify their effectiveness.
- We implement k-core decomposition algorithms on three representative GPU-based graph-parallel systems, Medusa [85], Gunrock [77] and GSWITCH [61].
- We compare the above algorithms and CPU-based k-core decomposition algorithms comprehensively using 20 public graph datasets of various characteristics.

In the rest of this paper, Section II briefly surveys existing parallel algorithms for k-core decomposition and other related works, and Section III reviews the important concepts in GPU programming. Then, Section IV describes our GPU-based peeling algorithm and its optimization techniques. Section V briefly discusses how to implement k-core decomposition on existing GPU graph-parallel systems. Finally, Section VI reports our experiments, and Section VII concludes this paper.

II. Existing Algorithms for k-Core Decomposition

This section reviews the related work. Specifically, Section II-A briefly surveys the existing algorithms for k-core decomposition including serial, parallel, distributed and GPU-based algorithms. Section II-B then briefly reviews the GPU-based graph-parallel systems which can be used to implement k-core decomposition algorithms. Finally, Section II-C reviews some other works related to k-core decomposition.

A. Existing Algorithms for k-Core Decomposition

BZ. BZ [30] is the state-of-the-art peeling algorithm for k-core decomposition, which runs in rounds. In the k^{th} round ($i = 0, 1, 2, \cdots$), BZ repeatedly removes a vertex with degree equal to k until no such vertex is left in G; the removed vertices have core value k and constitute a vertex set called the k-shell of G, denoted by $V^{(k)}$ hereafter. Let us denote the largest round number as k_{max}, after which no vertex is left. We call k_{max} as the core number of the graph G. To illustrate using Fig. 1, we have $k_{max} = 3$, and the 1-, 2- and 3-shells are colored in green, yellow and red, respectively. The k-core is given by $\cup_{i=k}^{k_{max}} V^{(i)}$, which are the subgraphs shown in Fig. 1 marked by dashed outlines. Note that in each round, the removal of a vertex may produce new vertices to remove, such as vertex A after the removal of vertex B when computing the 2-shell.

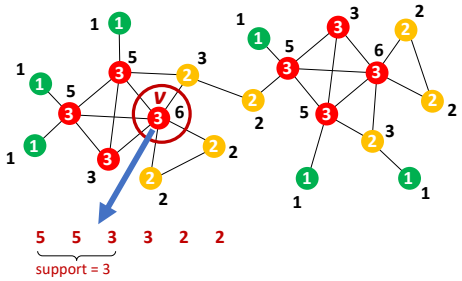

Fig. 2. Illustration of the h-Index Operator

The key contribution of BZ [30] is a smart linear-time implementation of the above idea with the help of four carefully selected arrays (see Section II-A of [41] for details).

Multicore Peeling Algorithms. ParK [41] is the first parallelization of the peeling algorithm. It adopts a two-phase approach to implement each peeling round k: (1) in the **scan** phase, the vertex degree array is scanned in parallel, and each thread collects its examined vertices with degree k into a global buffer B; (2) the **loop** phase where each thread keeps removing a vertex v from B, and adding those neighbors of v whose degree becomes k to B for future removal.

The loop phase of ParK is broken into sub-levels where each sub-level explores the neighbors of vertices in B to add new vertices with degree k into a new buffer B^{new}; at the end of the sub-level, we assign B^{new} to B to start a new breadth-first propagation in the next sub-level. Instead of using a global buffer B, PKC [51] removes the need of sub-level synchronization within the loop phase of each round, by letting each thread only access its local buffer B^{loc}. In the scan phase, each thread collects those vertices with degree k that it scans into B^{loc}. Then, each thread runs the loop phase independently by directly removing vertices from B^{loc} and appending new vertices to B^{loc}, so there is no sub-level synchronization.

Distributed Algorithms. MPM [63] is the pioneering distributed algorithms for k-core decomposition, followed by later works such as [79] for core maintenance on large dynamic graphs. In MPM, each vertex repeatedly performs h-index-style local updates to estimate its core number from the latest core-number estimates of its neighbors until convergence.

To illustrate using the graph G in Fig. 1, we plot the degree of each vertex near itself in Fig. 2. MPM initializes the core-number estimate of each vertex v, denoted by $a(v)$, as v's degree in G. Each vertex then repeatedly refines $a(v)$ by computing the h-index of the multiset $\mathcal{A} = \{a(u) \,|\, (u, v) \in E\}$, which (i) sorts $\mathcal{A}$'s elements in non-increasing order, and (ii) scans the sorted list from the beginning to find $\max\{i \in \mathbb{N}, \mathcal{A}[i] \geq i\}$ to update $a(v)$. For example, v in Fig. 2 has a sorted list of neighbor core-number estimates $\mathcal{A} = [5, 5, 3, 3, 2, 2]$. When $i = 1$ and 2, the element value $\mathcal{A}[i] = 5 > i$, so we continue to examine the 3rd element; since $\mathcal{A}[i] = 3 = i$, $\mathcal{A}[i] \geq i$ holds for the first time so

$a(v)$ is refined from the old value 6 to 3. Intuitively, v can find 3 neighbors with degree ≥ 3 but cannot find 4 neighbors with degree ≥ 4, so $core(v) \leq 3$. We use "$\leq$" here since $a(u)$ of these 3 neighbors u may further decrease when they are refined. When $a(v)$ converges for all $v \in V$, we have $core(v) = a(v)$ and k-core decomposition completes.

GPU Algorithms. We are only aware of two works studying GPU algorithms for k-core decomposition. The first is VETGA [60], which is currently the fastest GPU implementation. To utilize GPU, VETGA reframes the peeling algorithm in terms of vector primitives so that it can be executed using highly optimized GPU vector processing operations in PyTorch. The second work is [75] which peels from the highest core number to the lowest. This method needs to compute the graph core number for k_{max} times, and was found in [60] to be orders of magnitude slower than VETGA.

B. Graph-Parallel GPU Systems

Since the advent of Google's distributed system, Pregel [57], which promotes a think-like-a-vertex programming model and a bulk synchronous parallel (BSP) execution model, many vertex-centric systems have been developed [81] including GPU-based systems. Medusa [85] strictly mimics the vertex-centric BSP model of Pregel, where users may define a user-defined function (UDF) for a vertex v to send messages to its neighbors, and to receive a batch of messages for processing at v in the next iteration. Medusa can implement the h-index operator of MPM in the UDF for k-core decomposition.

Later GPU systems adopt a more restricted edge-centric UDF to enable more performance optimization, where given an edge (u, v), users define how the value of v is updated using the values of u and edge (u, v). While the edge-centric interface prevents us from implementing a MPM-style algorithm, we can still implement the peeling algorithm where when a vertex v deletes itself due to its degree $< k$, it can send a message to neighbors: the message along an edge (u, v) simply decrements the degree of v by 1. Systems following the edge-centric programming model include CuSha [54], MapGraph [45], Gunrock [77], Groute [31], Frog [72], Gluon [42], SEP-Graph [76], and GSWITCH [61].

McSherry et al. [59] noticed that existing graph-parallel systems add a lot of system-level overheads to the computation as compared to a direct implementation, but that study does not test the GPU systems. So, a goal of our current work is to compare the performance of our direct implementation of a peeling-based GPU algorithm with implementations of peeling- and MPM-based algorithms in GPU systems.

C. Other Algorithms and Problem Variants

Disk-based algorithms have been explored for k-core decomposition [35], [53], [78] to scale beyond the memory limit of a single PC, as well as streaming algorithms [68], [69].

Hierarchical core decomposition (HCD) of a graph G builds a forest structure where each tree node contains the vertices in a k-core connected component, and each tree edge represents that a k-core component contains another k'-core component with $k < k'$. HCD can be computed in linear time [58] and it can be used to find the best k-core component efficiently [37]. The parallel algorithm for HCD has been studied in [38].

Variants of k-core have also been studied, such as (k, h)-core [33], [40] which relaxes neighboring relationship to be within h hops, (k, r)-core [84] which adds an attribute-based pairwise vertex similarity constraint, and D-core [46], [47], [56] which extends the concept of k-core to directed graphs.

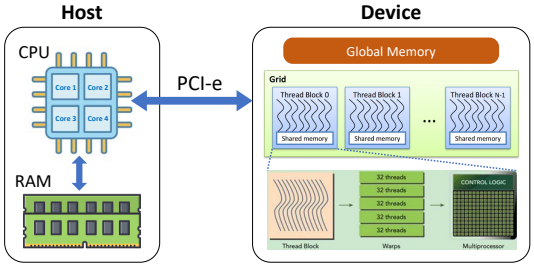

Fig. 3. GPU Architecture

III. GPU Preliminaries

This section briefly reviews the GPU architecture and CUDA programming, to prepare readers with minimal set of concepts necessary to understand our GPU algorithms.

Fig. 3 summarizes the GPU architecture. Specifically, a GPU device is connected to the host CPU via PCI-e bus. In NVIDIA CUDA architecture, developers write device programs called kernels. A kernel is usually explicitly configured and invoked by a CPU program to run on a GPU, with many threads running the same kernel program in parallel. A CPU program may call a serial of kernels for GPU execution, and each kernel (or, kernel grid) consists of an array of thread blocks that execute the same kernel program. Threads from the same block have access to low-latency shared memory and their execution can be synchronized. In contrast, different thread blocks are independent in their execution.

From the hardware perspective, a GPU device consists of an array of streaming multiprocessors (SMs), where each SM has a set of execution units and a chunk of shared memory. In an NVIDIA GPU, the basic unit of execution is warp. A warp is a collection of 32 threads that are executed simultaneously by an SM. Multiple warps can be executed on an SM at once. When an SM executes an SIMD instruction of a kernel program, it is executed on all threads. If different threads of an SM need to execute different control flows (e.g., different branches of an if-else block), the processor executes all paths, using masking to disable/enable the relevant threads as appropriate. As a result, a GPU program needs to be carefully designed to avoid path divergence that leads to GPU underutilization.

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM, and multiple thread blocks can execute concurrently on one SM. As thread blocks terminate, new blocks are launched on the vacated SMs.

Note that GPU threads cannot directly access the host memory so data movement is needed between the host memory

and GPU global memory before and after a kernel call. Due to the memory hierarchy of GPU and the intra-warp and inter-warp parallelism within an SM, coalesced memory accesses are preferred, and a GPU program should avoid irregular memory accesses. It is also favorable to keep the data that are frequently accessed by the threads of a block in the block's shared memory to reduce the memory latency as compared with directly accessing data from the global memory.

In a typical CUDA program, a CPU program first uses cudaMalloc(.) to allocate space for input data and intermediate processing buffers on the global memory of a GPU (called device memory hereafter), and uses cudaMemcpy(.) to move the input data from host memory to device memory. It then calls a series of kernel functions to let the GPU perform parallel computations, and finally obtains results from the device memory back to the host memory using cudaMemcpy(.), and frees the occupied device memory using cudaFree(.).

The name of a kernel function is specified in the form kernel_function<<<BLK_NUM, BLK_DIM>>>, where

- **kernel_function** is the function name of the kernel to launch with by the CPU program;
- **BLK_NUM** is the number of thread blocks to run by the kernel launch;
- **BLK_DIM** is the number of threads in each thread block.

Therefore, a kernel launch runs NUM_THREADS = BLK_NUM $\times$ BLK_DIM threads in total. All these threads run the same piece of code in serial as specified by the body of the kernel function, but on different data. This data parallelism is realized because each thread has access to the following variables properly configured by the kernel launch:

- **blockIdx.x:** the ID of a block, which takes a value in $\{0, 1, \cdots,$ BLK_NUM $-1\}$;
- **blockDim.x:** the number of threads in each block, aka. the block dimension, as specified by BLK_DIM;
- **threadIdx.x:** the ID of a thread in a block, which takes a value in $\{0, 1, \cdots,$ BLK_DIM $-1\}$.

The unique thread ID in a kernel grid can be obtained as

THREAD_ID = blockIdx.x * blockDim.x + threadIdx.x.

To process an array of n items where $n >$ NUM_THREADS, one can use the following for-loop:

for($i =$ THREAD_ID; $i < n$; i += NUM_THREADS)

so that, for example, the thread with THREAD_ID $= 2$ processes data items at positions 2, NUM_THREADS $+ 2$, $2 *$ NUM_THREADS $+ 2$, $\cdots$.

We have not seen the concept of warp yet, which is actually implicit: a CUDA programmer needs to be aware that the threads in each block is partitioned into warps of 32 threads (we assume BLK_DIM is always specified as a multiple of 32). For example, a thread can get its warp ID in a block as

WARP_ID = threadIdx.x / 32 (or threadIdx.x $>> 5$),

and the thread can get its ID in its warp as

LANE_ID = threadIdx.x % 32 (or threadIdx.x & 31).

They are very useful in our algorithm implementation in Section IV. For example, we can let Warp 0 of a block fetch data items needed for next iteration from global memory into the block's shared memory, while threads of the other warps compute data items of the current iteration. This method overlaps computation with IO and may reduce memory latency.

As another example, the processing of each vertex v is often done not by an individual thread, but rather by all threads in a warp in parallel where each thread processes one adjacency list item of v, in order to achieve coalesced memory access pattern of adjacency lists. Since the degree of v, denoted by $deg(v)$, can be > 32, the warp runs the following for-loop so that its threads process v's adjacency list in multiple iterations:

for($i =$ LANE_ID; $i < deg(v)$; i += 32),

where each thread processes the i^{th} item in the adjacency list. When $deg(v) < 32$, some threads in a warp are idle, leaving GPU cores underutilized. To address this problem, virtual warping [80] can be used to allow each physical warp to run 4 logical warps each with 8 threads, so that each virtual warp processes the adjacency list of an individual vertex. This technique is mainly for those graphs with a low average degree [80],and is orthogonal to our techniques in this work.

We next present our GPU-based peeling algorithms in Section IV, where we keep the narrative in a high level for readability, and the above programming details are implicit.

IV. Our Peeling Algorithms on a GPU

GPU memory is often the key restriction of the graph size that can be processed by a GPU algorithm, so it is important to ensure that a graph is stored compactly in the global memory.

Graph Organization in GPU. We keep a graph $G = (V, E)$ in the global memory compactly as three arrays:

- ***neighbors***: the concatenation of the adjacency lists (i.e., neighbor ID lists) of all vertices in V;
- ***offset***: *offset*$[i]$ = the start location of the neighbor list of Vertex i in *neighbors*;
- ***deg***: *deg*$[i]$ = the degree of Vertex i.

Here, we assume the vertex IDs are densely indexed; if they are not, we can perform ID recoding [82] of G as preprocessing. These three arrays are precomputed by the CPU program and moved to the device memory before kernel launches.

With these arrays, we can obtain the neighbors of Vertex i as *neighbors*[*offset*$[i] + j$] where $j = 0, 1, \cdots,$ *deg*$[i] - 1$. This is a consecutive subarray that allows coalesced memory access.

In the sequel, we first describe our peeling-based GPU algorithm, and then explore a few optimizations techniques.

A. Our Basic GPU Algorithm: Overview and Challenges

Our algorithm follows the two-phase algorithm framework of PKC [51] as reviewed in Section II-A, but it is non-trivial to implement the two-phase algorithm in a GPU setting.

We identify 3 challenges. The **first challenge** is to determine the proper the smallest computing unit for parallelism. In PKC [51], that unit is a CPU thread, but a thread is clearly a bad choice in our GPU setting since for coalesced memory access of adjacency lists, each vertex is examined by a warp (with 32 threads). This leaves us two options, warp or block.

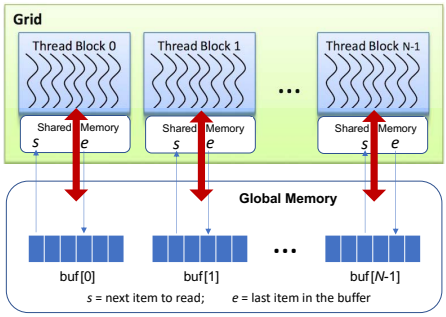

Fig. 4. Vertex Buffers for the Thread Blocks

Recall that each thread in PKC [51] is associated with a local buffer B^{loc}, and the local buffers evenly partition the host memory space. In our GPU setting, each computing unit needs to have a local buffer in the global memory of GPU. Since there are many warps, if warp is the computing unit, its local buffer has to be small. Since different warps may collect imbalanced numbers of initial degree-k vertices in the scan phase of iteration k, a small buffer size is more prone to overflow. We, therefore, use block as the computing unit. Accordingly, the remaining global memory (except the space occupied to store G) is evenly partitioned into BLK_NUM arrays, where block i is associated with an array as its buffer, denoted by *buf*$[i]$, and all the BLK_DIM threads of a block read and write the same buffer. This is illustrated in Fig. 4.

Our two-phase algorithm is implemented by calling two kernels in each peeling round k: (1) in the scan-phase kernel, vertices are partitioned among all the blocks, and block i scans its assigned portion of vertices and collects those with degree k into *buf*$[i]$; (2) in the loop-phase kernel, each warp of block i fetches a degree-k vertex v from *buf*$[i]$, deducts 1 from the degree of its neighbors with degree $> k$, and then adds all degree-k neighbors to *buf*$[i]$ for further k-shell propagation.

Intuitively, in the loop phase, the blocks conduct BFS to reach all the k-shell vertices in parallel from their respective initial k-shell vertices collected by the scan phase.

Here, two more challenges exist in implementation. The **second challenge** is that, a vertex in the k-shell could be a neighbor of different vertices traversed by different blocks, and conflict resolution is needed to ensure that it is only collected by one of the blocks to avoid redundant computation. We will explain our solution in Section IV-B when describing the loop kernel (Algorithm 3), with an illustration using Fig. 6.

The **third challenge** is that, all threads of block i are fetching vertices from and adding vertices to the same buffer *buf*$[i]$, and mechanism is needed to ensure thread-safety of buffer updates. Specifically, we maintain two positions s and e for each *buf*$[i]$ in the shared memory of block i for thread-safe access by all its threads (see Fig. 4), where s denotes the position of the next element in *buf*$[i]$ to process, and e denotes the next position in *buf*$[i]$ to append an element (i.e., one position after the last element currently in *buf*$[i]$).

B. Details of Our Basic GPU Algorithm

The Host Program. Algorithm 1 shows our host program. Specifically, Line 1 first loads the input graph into the device memory including the three arrays *neighbors*[.], *offset*[.] and *deg*[.] introduced at the beginning of Section IV. Note that our algorithm will update *deg*[.] when vertices in k-shells are being removed so that in the end, for every vertex $v \in V$, $deg[v]$ keeps the value of $core(v)$ (c.f. Line 10). Then, Line 2 initializes $count$ to 0, which is a counter indicating how many vertices have already been removed during the course of algorithm execution. Since vertex examinations are conducted on the GPU, we need to mirror $count$ to the device memory as gpu_count in Line 4 so that GPU threads can update it. Before beginning the k-shell removal for $k = 0, 1, 2, \ldots$ by the while-loop in Lines 5–9, Line 4 initializes k as 0 and allocates space of the buffers for all thread blocks in the device memory.

Algorithm 1 GPU-Based Peeling Algorithm: Host Program

Input: $G = (V, E)$
Output: Core value $core(v)$ for every vertex $v \in V$

1: Load G into the device memory, including *deg*[.]
2: Set vertex counter $count \leftarrow 0$
3: Load $count$ to device memory as gpu_count
4: $k \leftarrow 0$; Allocate buffers *buf*[.][.] in device memory
5: **while** $count < |V|$ **do**
6: Launch kernel *scan*(k)
7: Launch kernel *loop*(k)
8: Read gpu_count back to $count$
9: $k \leftarrow k + 1$
10: Read *deg*[.] back to host memory and return as core values

The k-shell removal is repeated until all $|V|$ vertices have been removed (see Line 5). Specifically, we call the kernel *scan*(k) in Line 6 to let thread blocks collect their initial sets of k-shell vertices into their buffers, which are then used by the other kernel *loop*(k) in Line 7 to collect and remove the remaining vertices in the k-shell using parallel BFS. These two kernels are described in Algorithms 2 and 3, respectively.

The Scan-Phase Kernel. Algorithm 2 shows the scan-phase kernel, where the vertices of G are distributed to the threads for checking their degrees. Lines 3–5 ensure that every vertex is assigned to a unique thread. For example, Thread j processes all vertices with IDs of the form $n *$ NUM_THREADS $+ j$ ($n \in \mathbb{N}$) is processed by Thread n_2. If vertex v has degree k (Line 6), it must be in the k-shell (since k'-shells with $k' < k$ have been removed in previous rounds), so Lines 7 and 9 append v to the buffer of the current thread's thread

Algorithm 2 Kernel Function *scan*(k)

Assumption: e is in shared memory

1: **if** THREAD_ID $= 0$ **do** $e \leftarrow 0$
2: __syncthreads()
3: **for** ($s \leftarrow 0$; $s < |V|$; s += NUM_THREADS) **do**
4: $v \leftarrow s +$ THREAD_ID
5: **if** $v \geq |V|$ **do continue**
6: **if** $deg[v] = k$ **then**
7: $pos \leftarrow$ atomicAdd(e, 1)
8: {Let the thread block of the current thread be i}
9: *buf*$[i][pos] \leftarrow v$

block. Here, CUDA function atomicAdd(e, 1) returns pos as the current value of e (i.e., the next position to append a new buffer element), and it advances e atomically so that other threads later can only write to buffer positions after pos.

Recall from Fig. 4 that e is shared by all threads of a block and is kept in the block's shared memory. It is initialized by Thread 0 of the block in Line 1, and a block-level synchronization barrier in Line 2 then ensures that no update of e in Line 7 can be executed before Line 1 of Thread 0.

While a block contains many threads that update e, shared memory atomic operations have been highly optimized by NVIDIA with native hardware support [64].

At the end of kernel $scan(k)$, Thread 0 of each block also needs to back up e from its shared memory to $buf[i].e$ in the global memory for use by the second kernel $loop(k)$, which is omitted in Algorithm 2 to make it succinct.

The Loop-Phase Kernel. Algorithm 3 shows the loop-phase kernel. Specifically, each block i first initializes its buffer head and tail positions, s and e (both in shared memory), respectively, in Line 2. Here, (1) e basically loads $buf[i].e$ in the global memory previously written by $scan(k)$; (2) this is done only by Thread 0 of each block (see Line 1); (3) Line 4 ensures initialized s and e to be seen by all threads of a block.

Afterwards, each block i repeatedly fetches k-shell vertices from *buf*$[i]$ for processing (Lines 4–24) until there are no more vertices in *buf*$[i]$, i.e., $s = e$ as checked in Line 5. Recall that

Algorithm 3 Kernel Function $loop(k)$

Assumption: s and e are in shared memory

```
1:  if THREAD_ID = 0 then
2:      s ← 0,  e ← buf[i].e
3:  repeat
4:      __syncthreads()
5:      if s = e do break
6:      s' ← s + WARP_ID,  e' ← e
7:      __syncthreads()
8:      if s' ≥ e' do continue
9:      if THREAD_ID = 0 do
10:         s ← min{s + BLK_DIM >> 5, e}
11:     {Let the thread block of the current thread be i}
12:     v ← buf[i][s']
13:     pos_s = offset[v],  pos_e = offset[v + 1]
14:     repeat
15:         __syncwarp()
16:         if pos_s ≥ pos_e do break
17:         pos ← pos_s + LANE_ID,   pos_s ← pos_s + 32
18:         if pos ≥ pos_e do continue
19:         u ← neighbors[pos]
20:         if deg[u] > k do
21:             deg_u ← atomicSub(deg[u], 1)
22:             if deg_u = k + 1 do
23:                 loc ← atomicAdd(e, 1),  buf[i][loc] ← u
24:             if deg_u ≤ k do atomicAdd(deg[u], 1)
25: __syncthreads()
26: if THREAD_ID = 0 do atomicAdd(gpu_count, e)
```

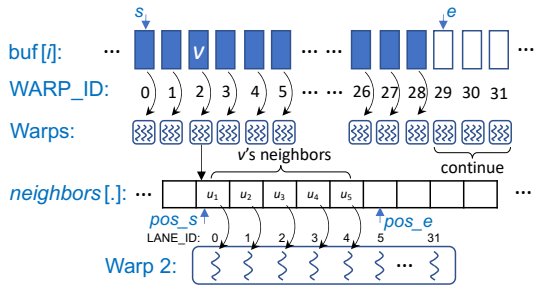

Fig. 5. Illustration of Algorithm 3 (Assuming a Block has 32 Warps)

(1) each block has BLK_DIM/32 warps, and (2) each vertex v (i.e., its neighbor list) is processed by a warp. Therefore, in each iteration, a block processes BLK_DIM/32 vertices.

We break each iteration into 3 parts: (i) Lines 4–8 determine if the loop phase of the current block i terminates; (ii) Lines 9–13 advance to the next batch of vertices in *buf*$[i]$, and retrieve from *buf*$[i]$ the vertex v that is assigned to the current warp; and (iii) Lines 14–24 let the 32 threads of the current warp process v's neighbors u in parallel, where u is added to *buf*$[i]$ if its degree becomes k after the removal of v from G.

First consider **Part (i)** in Lines 4–8. Let the block of the current thread be block i. Due to the block-level synchronization barrier in Line 4, all threads of a block will see the same value of s in Line 5, which equals the next location of *buf*$[i]$; the warps of block i will retrieve vertices *buf*$[i][s]$, *buf*$[i][s+1]$, *buf*$[i][s+2]$, $\cdots$ for processing, i.e., v in Line 12 where $s' = s+$ WARP_ID was set in Line 6. Fig. 5 shows how elements in *buf*$[i]$ are assigned to the warps of block i. Note that s was advanced in the previous iteration by Line 10.

Now let us return back to Line 5: if $s = e$, then *buf*$[i]$ is empty and all threads of block i are finished and thus break out of the loop. Otherwise, Line 6 obtains the vertex position s' in *buf*$[i]$ that the warp of the current thread should process; it also backs up the buffer tail e after the last iteration into e', so that if $s' \geq e'$ (i.e., block i has more warps than the remaining vertices in *buf*$[i]$) in Line 8, $v =$ *buf*$[i][s']$ does not exist so the current warp skips Parts (ii) and (iii). For example, in Fig. 5, $e' \leftarrow e = s + 29$ in Line 6, and since $s' = s +$ WARP_ID $\geq e'$ for Warps 29, 30, and 31, these three threads execute "continue". Note that "continue" is used rather than "break" here, since the warp needs to stay active to process the next iteration as more vertices could be appended to *buf*$[i]$ in the current iteration. For example, in Fig. 5, Warp 2 processes vertex v, so its threads will examine v's neighbors u_1, ..., u_5 in Part (iii) and may add up to 5 vertices into *buf*$[i]$. Since each vertex in this iteration may add multiple vertices to *buf*$[i]$, there could be many new k-shell vertices in *buf*$[i]$ to keep all warps of block i busy in the next iteration.

The block-level barrier in Line 7 is needed, since warps of a block may run in any order; without the barrier, there is no guarantee that the read of s and e in Line 6 is executed strictly before any updates of s and e in Lines 10 and 23, respectively. Moreover, we cannot swap Lines 7 and 8 since otherwise, when some warps cannot find a vertex in *buf*$[i]$ to process, "continue" is called so they will never run __syncthreads() to

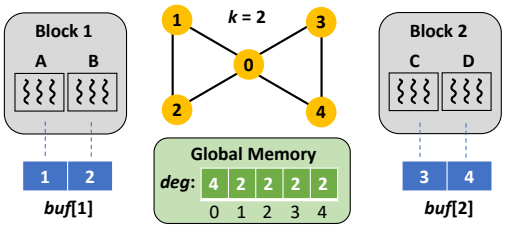

Fig. 6. Redundancy Avoidance and Core Number Maintenance

advance the barrier, causing the kernel program to get stuck.

Next consider **Part (ii)** in Lines 9–13. Since s' has been computed in Line 6 for each warp to retrieve vertex $buf[i][s']$ in Line 12, s is no longer used in the rest of the current iteration, and thus advanced by Thread 0 of block i in Line 10 for use by the next iteration. For example, in Fig. 5, $s \leftarrow \min\{s+32, e\} = \min\{s+32, s+29\} = e$, which is right after the last element of $buf[i]$ for the current iteration. Afterwards, Line 12 retrieves $v = buf[i][s']$ and Line 13 retrieves its adjacency list for use in Part (iii). Fig. 5 shows how pos_s and pos_e are set by Line 13 for vertex v processed by Warp 2.

Finally consider **Part (iii)** in Lines 14–24, where the 32 threads of a warp process up to 32 neighbors of v in each iteration of the inner-loop given by Line 14, until all neighbors of v are processed. This latter condition is examined in Line 16, and to ensure that all 32 threads see the same values of pos_s and pos_e, a warp-level barrier is added in Line 15. As Lines 17 and 19 show, the j^{th} thread of a warp examines v's neighbor $u = neighbors[pos\_s+j]$, and pos_s is advanced by 32 for use by the next inner-iteration. For example, in Fig. 5, the first five threads of Warp 2 process v's five neighbors u_1–u_5, respectively. Here, pos_s is a local variable and each thread of a warp has a replica. It is possible that there are fewer than 32 neighbor items left in the neighbor list of v, in which case Line 18 skips the execution of those threads without a corresponding u to examine. For example, in Fig. 5, the thread of Warp 2 with LANE_ID $= 5$ finds that $pos = pos\_s + 5 \geq pos\_e$, so Line 18 executes "continue".

Each thread then processes a neighbor u of v in Lines 20–24. Specifically, the removal of v reduces u's degree by 1, so Line 20 skips u if its degree $\leq k$ since such a u belongs to k'-shell for some $k' < k$, and thus $deg[u]$ should have converged to $core(u)$ in a previous round. Otherwise, Line 21 decrements $deg[u]$ to reflect the removal of v. CUDA's atomicSub(.) in Line 21 returns the old value of $deg[u]$ before its decrement, so if the returned value $deg\_u = k+1$ in Line 22, $deg[u]$ has been decremented to k by Line 21. Thus, u belongs to the k-shell and is hence appended to $buf[i]$ in Line 23.

Finally in Line 26, each block adds its collected number of k-shell vertices in this round (i.e., e) to *gpu_count*. Recall that Algorithm 1 relies on *gpu_count* to decide the program termination condition (see Lines 5 and 8 of Algorithm 1).

Avoiding Redundant Vertex Traversal. One issue remains (i.e., the second challenge introduced in Section IV-A) as is illustrated in Fig. 6, where we consider the toy example of two blocks each with two warps, and assume that (1) we are now in the loop phase for finding 2-shell, and that (2) in the scan phase, $buf[1]$ has collected degree-2 vertices 1 and 2, while $buf[2]$ has collected vertices 3 and 4. Since Warps A ($v = 1$), B ($v = 2$), C ($v = 3$) and D ($v = 4$) will all see neighbor $u = 0$ in Line 19, assuming that their four respective threads processing 0 reach Line 20 simultaneously and find $deg[0] = 4 > 2$, then they will all run Line 21 to decrement $deg[0]$ from 4 all the way to 0. Since each thread decrements $deg[0]$ atomically, the thread that executes Line 21 the second will find $deg\_u = 3$ (i.e., $deg[u] = 2$) and add vertex 0 to its block's buffer; note that a k-shell vertex will only be captured by one block so there is no redundant computation.

After all the 4 threads complete Line 21 simultaneously, $deg[0]$ is reduced to 0 but the core number of vertex 0 is 2. To allow $deg[0]$ to converge correctly to $core(0) = 2$, we thus need to have Line 24, which will be executed twice by the 2 threads seeing $deg\_u = 1$ and 2 (i.e., $deg[u] = 0$ and 1), respectively, to recover $deg[u]$ back to 2.

In general, **Case 1:** if u is in k-shell, then if any k-shell vertex v causes $deg[u]$ to be decremented below k, this decrement will be canceled by Line 24, and since u has at least k such k-shell neighbors v, $deg[u]$ will be added back to k to reach the correct $core(u)$. **Case 2:** if u is in k'-shell with $k' > k$, then $deg\_u \geq k$ in Line 24 cannot happen since u has at least k' neighbors in k'-shell, and this round simply reduces $deg[u]$ by the number of neighbors in k-shell by Line 21 for correct peeling. **Case 3:** if u is in k'-shell with $k' < k$, then $deg[u] > k$ in Line 20 cannot hold, so $deg[u]$ will not be updated in this round (and also in future rounds by induction), i.e., $core(u)$ has converged to k'.

C. Optimization Techniques

We explore a few optimization techniques to our algorithm.

Ring Buffers. In Fig. 4, the buffer of each block is organized as an array of fixed size (let the size be B). There are two problems: (1) if e reaches B (which can be checked by an assert statement), we have a block overflow so that the graph is too large to be processed given the space limit of the global memory; (2) since s is incremented in one direction, buffer slots before position s cannot be recycled and are thus wasted. To address these problems, we can organize each buffer $buf[i]$ as a ring buffer, where we update Line 23 of Algorithm 3 with $buf[i][loc \mod B] \leftarrow u$ so that if e is incremented to B or beyond, it wraps around from the beginning of the buffer array. Similarly, we set $v \leftarrow buf[i][s' \mod B]$ in Line 12.

Utilizing Shared Memory for Buffering. While we have used the shared memory of each block i to keep s and e of its buffer $buf[i]$, buffer elements are still read from and written to the global memory: see Line 9 of Algorithm 2, and Lines 12 and 23 of Algorithm 3. A natural idea is to use the remaining space in the shared memory of a block as a low-latency buffer, denoted by $\mathcal{B}$. We use $\mathcal{B}$ only in the loop phase, not the scan phase since otherwise, we would still need to flush its content to $buf[i]$ in global memory for use by the "loop" kernel, which causes additional overheads. Ideally, when the k-shell is small enough, a block i in the loop phase should only add new vertex

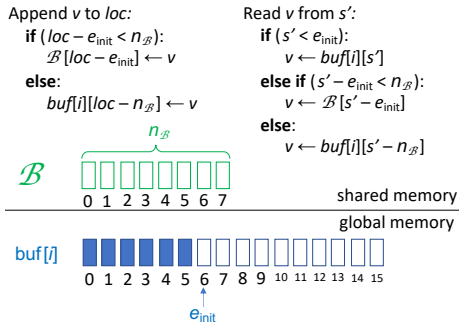

Fig. 7. Vertex Buffering in Shared Memory

to $\mathcal{B}$ for k-shell propagation, though the initial set of k-shell vertices are read from *buf*$[i]$ as collected by the scan kernel.

Fig. 7 illustrates this idea, where we assume the capacity of $\mathcal{B}$ is $n_{\mathcal{B}} = 8$, and initially, block i has already collected $e_{init} = 6$ k-shell vertices into *buf*$[i]$ during the scan phase. In Line 12, we now need to translate s' into a position in either $\mathcal{B}$ or *buf*$[i]$ when fetching a vertex, which is illustrated in Fig. 7. For example, if $s' = 3$, we read *buf*$[i][3]$. As soon as the initial k-shell vertices in *buf*$[i]$ have all been fetched, we start to read from $\mathcal{B}$, e.g., if $s' = 7$, we read $\mathcal{B}[7 - e_{init}] = \mathcal{B}[1]$. When $\mathcal{B}$ has been exhausted, we read again from *buf*$[i]$, e.g., if $s' = 14$, we read *buf*$[i][14 - n_{\mathcal{B}}] =$ *buf*$[i][6]$.

Similarly, in Line 23, we now translate loc into a position in either $\mathcal{B}$ or *buf*$[i]$ when appending a vertex, which is also illustrated in Fig. 7. For example, if $loc = 7$, we write $\mathcal{B}[1]$. When $\mathcal{B}$ has been exhausted, we then continue to append to *buf*$[i]$, e.g., if $loc = 14$, we write *buf*$[i][6]$. Note that the position translation by all threads in block i needs to access e_{init}, so we let Thread 0 read it into the shared memory when the "loop" kernel begins.

Vertex Frontier Prefetching. In our kernel implementation, we adopt BLK_DIM $= 1024$, so each block has BLK_DIM $>> 5 = 32$ warps, equaling the number of threads in a warp!

With this setting, we can implement a prefetching strategy where Warp 0 of each block i fetches 31 k-shell vertices at the propagation frontier (i.e., v in Line 12) into the shared memory, for the remaining 31 warps to process in the next iteration; while in the meanwhile, the other 31 warps are processing the 31 vertices fetched by Warp 0 in the previous iteration, without reading from *buf*$[i]$ in the global memory.

To implement this logic, we maintain a small vertex array *pref*$[.]$ in the shared memory where *pref*$[j]$ keeps the prefetched vertex for Warp j. In Algorithm 3, the Thread-0 logic in Lines 9–10 is now replaced by Warp-0 logic, where (1) the thread with LANE_ID $= 0$ first advances s, (2) followed by __syncwarp() to ensure other threads in Warp 0 sees the advanced s, then (3) the remaining 31 threads fetch the next batch of up to 31 k-shell vertices into *pref*$[.]$, where each thread fetches *buf*$[i][s' - 1]$ into *pref*$[LANE\_ID]$, where s' was computed in Line 6.

Reducing Contention for Buffer Appending. In Line 7 of Algorithm 2 and Line 23 of Algorithm 3, all the 1024 threads of each block (atomically) increments the same variable e

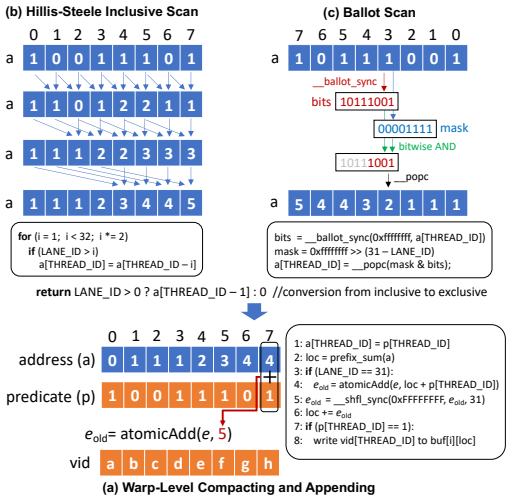

Fig. 8. Illustration of Compacting and Appending

in the shared memory, so the contention is high. In each iteration, a thread only examines at most one vertex v (in the scan kernel) or one neighbor u (in the loop kernel) in each iteration, so each warp (or block) collects at most 32 (or 1024) elements. Let the number of collected items be m, then we let a thread of the warp (or block) advance e by m positions in one atomicAdd(.) operation, and then let all the threads write these m elements to the m advanced positions in *buf*$[i]$.

This can be achieved using the "compact" operation with the help of the "scan" (or, prefix sum) operation, as illustrated by Fig. 8 where we assume warp-level compaction. To save figure space, we show a warp as having 8 threads instead of 32, and array indices are marked with local ones in a warp.

As Fig. 8(a) shows, each block i keeps three arrays of size BLK_DIM in shared memory: (1) $vid[j]$ tracks the ID of the vertex examined by Thread j; (2) $p[j]$ is set to 1 if Thread j finds vertex $vid[j]$ to be in k-shell (for *buf*$[i]$ insertion), and 0 otherwise; (3) $a[j]$ tracks the number of 1's in $p[.]$ before position j and inside Thread j's warp. For example, in Fig. 8(a), the first thread checks vertex a and finds that it should be appended to *buf*$[i]$, so $p[0] = 1$. The second thread checks vertex b and decides not to append it to *buf*$[i]$, so $p[1] = 0$. Also, $a[5] = 3$ since before position 5 there are 3 ones: $p[0]$, $p[3]$ and $p[4]$. We will explain how to compute $a[.]$ from $p[.]$ in the next paragraph using prefix sum (Lines 1–2). Note that the # of elements to insert for a warp equals the sum of the last elements of $a[.]$ and $p[.]$ in the warp, which is summed by the warp's last thread in Lines 3–4. For example, $a[7] = 4$ since the first 7 threads set 4 ones in $p[.]$, and as $p[7]$ is also one, we advance e by 5. So far, only the last thread in the warp gets e_{old} for the warp to start writing the 5 elements from, so Line 5 broadcasts it to the other 31 threads in the warp using a CUDA warp-level primitive. Each thread then gets its write location in Line 6, and writes its checked k-shell vertex to *buf*$[i]$ in Lines 7–8. For example, vertices a, d, e, f and h are written to locations e_{old}, $e_{old} + 1$, $e_{old} + 2$, $e_{old} + 3$, $e_{old} + 4$, respectively. Here, locations are computed as $e_{old} + a[j]$, $j = 0, 3, 4, 5, 7$ where $p[j] = 1$.

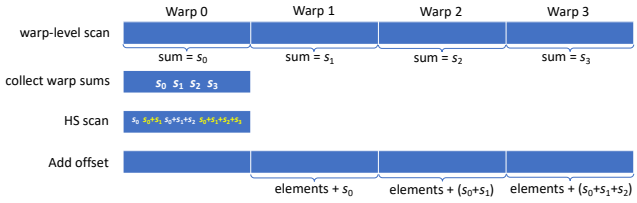

Fig. 9. Intra-Block Scan

We now explain how the warp-level scan in Line 2 of Fig. 8(a) is implemented. A straightforward approach is to use a logarithmic-span scan algorithm such as Hillis and Steele (HS) algorithm [50] and Blelloch algorithm [32]. We adopt HS as illustrated in Fig. 8(b) which runs for $\log n$ iterations since Blelloch algorithm needs twice the number of iterations. As shown in Fig. 8(b), in the i^{th} iteration, Thread j sums $a[j]$ with $a[j-2^{i-1}]$ if it is within the warp, so for example, a[7] counts the number of 1's from positions 6, 4, 0 to the current position 7 in iterations 1, 2, 3, respectively. Note that HS is inclusive meaning that the final prefix count includes the current position, so we need to compute exclusive counts (see the blue arrow in Fig. 8) for use by "compact" operation.

We also explore the more efficient Ballot scan algorithm [29] designed specifically for the warp-level scan, as illustrated in Fig. 8(c). Specifically, all threads of a warp first call CUDA warp-level primitive __ballot_sync to compact their 0-1 values into a 32-bit bitmap $bits$. Then, each thread j computes a mask to fetch the last j bits of $bits$, and counts the number of 1's therein using CUDA's __popc function.

Warp-level compacting can be applied in both the scan kernel and the loop kernel. We can also adopt block-level compacting in the scan kernel, but not the loop kernel since it operates in warp level where all threads of a warp check the neighbors of a vertex u simultaneously. While we can directly use HS for intra-block scan, it is more efficient to use the two-stage algorithm of [71] as illustrated in Fig. 9. Specifically, (1) each warp first computes the warp-local offsets using HS scan; (2) recall that we can obtain the number of 1's in each warp from the last element, so these sums are collected to Warp 0; (3) there are 1024 threads (32 warps) in a block and thus 32 sums are collected by Warp 0, which runs HS to compute their prefix sums (we cannot use ballot scan here since elements are not 0-1); (4) the offsets in each warp add the global offset computed by Warp 0, to get the block-level offsets for block-level compact. This approach has overheads since a block-level barrier is needed between two consecutive steps, and only Warp 0 computes in Steps (2) and (3).

V. Algorithms on Graph-Parallel Systems

We study k-core decomposition algorithms on three representative GPU-based graph-parallel systems, Medusa [85], Gunrock [77] and GSWITCH [61]. We choose Medusa [85] since it strictly mimics the model of Pregel where a vertex has access to the messages/values of all its neighbors in a UDF so that the h-index operator can be supported to implement an MPM-style algorithm in addition to the peeling algorithm. Gunrock [77] is selected since it features a then novel data-centric abstraction centered on operations on a vertex or edge frontier, and is known for its high performance. GSWITCH [61] is selected as a very recent system that supports algorithmic autotuning which configures a GPU kernel of a computing iteration for favorable performance based on features observed from the previous iteration. These three systems were proposed in 2014, 2016 and 2019, respectively, spanning evenly in the entire period of active development of GPU-based graph-parallel systems and are hence representative.

MPM-Style Algorithm on Medusa. A Medusa program requires users to implement 3 UDFs, (1) SendMessage, (2) CombineMessage, and (3) UpdateVertex. To implement MPM, "SendMessage" simply sends the core-number estimate of the current vertex to all its neighbors; "CombineMessage" implements the h-index operator over the received core-number estimates from all neighbors; "UpdateVertex" simply updates the core-number estimate of the current vertex using the new h-index value, and if the estimate value changes, a global aggregate flag is set to indicate more iterations are needed.

Peeling Algorithm on Medusa. In our implementation, "SendMessage" checks if the current vertex v is marked as deleted or if it has degree $> k$. If so, v sends 0 to all neighbors, while otherwise, v is in the k-shell, so we set $core(v) = k$, mark v as deleted, and send 1 to all neighbors. "CombineMessage" simply sums all messages received at a vertex v, which counts the number of deleted neighbors of v in the k-shell. "UpdateVertex" deducts the degree of a vertex v by this count, and if the resulting degree $\leq k$, we set a global aggregate flag to indicate that the current k-shell round should run more iterations. A typical Medusa program only has one level of iteration looping. We further add an outer loop of rounds, which terminates if all vertices are marked deleted.

Peeling Algorithm on Gunrock and GSWITCH. We directly use the k-core decomposition algorithm already implemented in Gunrock. Its idea is similar to our GSWITCH program to be described next. Specifically, a GSWITCH program requires users to implement 2 key UDFs, (1) "filter" which identifies new vertices with degree k to emit messages, and (2) "comp" which decrements the degree of each vertex for each received message. GSWITCH also provides a UDF "emit" for aggregating vertex states to indicate if the inner loop of iterations for a round k still needs computation. However, GSWITCH does not support an easy way to write the outer loop of rounds, so we simply repeat the iterative computations for n rounds, where n is hardcoded as the core number of each input graph.

VI. Experiments

We report our empirical results of the serial, CPU-parallel and GPU-parallel k-core decomposition programs in this section. Our CPU programs were run on a server with two Intel Xeon E5-2680 v4 CPU @ 2.40 GHz (48 threads) and 256 GB RAM. Our GPU kernels were run with NVIDIA Tesla P100 GPU with a global memory of 16 GB.

We find that different runs of our GPU programs may vary sometimes by over 30% in time, so we run each program

TABLE I
DATASETS

Dataset	$\|V\|$	$\|E\|$	d_{avg}	std	d_{max}	k_{max}	Category
amazon0601	403,394	3,387,388	16.8	15	2,752	10	Co-purchasing
wiki-Talk	2,394,385	5,021,410	4.2	103	100,029	131	Communication
web-Google	875,713	5,105,039	11.7	39	6,332	44	Web Graph
web-BerkStan	685,230	7,600,595	22.2	285	84,230	201	Web Graph
as-Skitter	1,696,415	11,095,298	13.1	137	35,455	111	Internet Topology
patentcite	3,774,768	16,518,948	8.8	10	793	64	Citation Network
in-2004	1,382,908	16,917,053	24.5	147	21,869	488	Web Graph
dblp-author	5,624,219	24,564,102	8.7	11	1,389	14	Collaboration
wb-edu	9,845,725	57,156,537	11.6	49	25,781	448	Web Graph
soc-LiveJournal1	4,847,571	68,993,773	28.5	52	20,333	372	Social Network
wikipedia-link-de	3,603,726	96,865,851	53.8	498	434,234	837	Web Graph
hollywood-2009	1,139,905	113,891,327	199.8	272	11,467	2,208	Collaboration
com-Orkut	3,072,441	117,185,083	76.3	155	33,313	253	Social Network
trackers	27,665,730	140,613,762	10.2	2,774	11,571,953	438	Web Graph
indochina-2004	7,414,866	194,109,311	52.4	391	256,425	6,869	Web Graph
uk-2002	18,520,486	298,113,762	32.2	145	194,955	943	Web Graph
arabic-2005	22,744,080	639,999,458	56.3	555	575,628	3,247	Web Graph
uk-2005	39,459,925	936,364,282	47.5	1,536	1,776,858	588	Web Graph
webbase-2001	118,142,155	1,019,903,190	17.3	76	263,176	1,506	Web Graph
it-2004	41,291,594	1,150,725,436	55.7	883	1,326,744	3,224	Web Graph

for 100 times and report the average and standard deviation (std). For all the other programs, each reported experiment was repeated for 3 times with the average reported.

We configure a kernel grid to have BLK_NUM $= 108$ blocks, each with BLK_DIM $= 1024$ threads (or, 32 warps). The global-memory buffer capacity of each thread block (i.e., capacity of $buf[i]$) is set to hold 1 million vertex IDs. Each block also has a shared-memory buffer $\mathcal{B}$ with a capacity of 10,000 vertex IDs, close to the space limit of a shared memory.

Datasets. We test our GPU algorithm and the other baselines extensively on 20 public graph datasets of various size, density, and core number, as shown in Table I where the datasets are listed in ascending order of the number of edges. These datasets are from different categories including (1) web graphs web-Google [23], web-BerkStan [24], in-2004 [21], wb-edu [10], wikipedia-link-de [25], trackers [18], indochina-2004 [2], uk-2002 [1], arabic-2005 [4], uk-2005 [5], webbase-2001 [16], it-2004 [3]; (2) interaction networks such as communication network *wiki-Talk* [26], citation network patentcite [8], collaboration networks dblp-author [9] and hollywood-2009 [13], and social networks oc-LiveJournal1 [17] and com-Orkut [19]; (3) co-purchasing network *amazon0601* [6]; (4) Internet topology as-Skitter [14]. These datasets are widely used in prior works on serial and parallel k-core decomposition. Some graphs are directed and we make them undirected by ignoring the edge direction.

As we can see, the graphs have very different characteristics. The number of edges is up to 1.15 billion on *it-2004*. The average (resp. max) vertex degree is up to 199.8 on *hollywood-2009* (resp. 11.57 million on *trackers*). The core number (i.e., k_{max}, the largest k for a k-core) is up to 6,869 on *indochina-2004* meaning that peeling-based computation needs to run for 6,870 rounds. The degree distribution can be very biased with a standard deviation of 2774 on *trackers* while its average degree is merely 10.2. Our GPU program can handle all these graphs since our graph is stored compactly in the global memory.

Ablation Study. We first compare our basic GPU algorithm described in Section IV-B, denoted by *Ours*, with its variants that integrate the various optimization techniques described in Section IV-C. The first two techniques concern how to reduce the memory latency of reading vertices from $buf[i]$:

- **SM** (Shared-Memory Buffering) uses a shared-memory buffer $\mathcal{B}$ to reduce the cost of accessing $buf[i]$ (c.f. Fig. 7).
- **VP** (Vertex Frontier Prefetching) lets Warp 0 of a block fetch k-shell vertices from $buf[i]$ into shared memory.

The next two techniques concern how new k-shell vertices are collected and appended to $buf[i]$, instead of appending them to $buf[i]$ one at a time using atomicAdd(.) as in *Ours*:

- **BC** (Ballot Compaction), for both scan and loop kernels, conducts warp-level compacting first as illustrated in Fig. 8 and then appends to $buf[i]$ in one batch.
- **EC** (Efficient Compaction) conducts block-level compacting in scan kernel (c.f. Fig. 9) to append to $buf[i]$ in one batch, and warp-level compacting in loop kernel.

TABLE II
ABLATION STUDY (TIME UNIT: MILLISECOND; AVG ± STD)

Dataset	Ours	SM	VP	BC	BC+SM	BC+VP	EC	EC+SM	EC+VP
amazon0601	**1*±0.2**	1±0.0	1±4.1	1±0.0	1±0.0	2±0.0	1±0.5	2±0.0	2±0.0
wiki-Talk	**36*±0.5**	38±0.0	38±0.0	45±0.4	45±0.1	48±0.3	59±0.3	60±0.0	60±0.1
web-Google	**5*±0.2**	6±0.0	6±0.0	6±0.0	7±0.0	8±0.0	8±0.0	9±0.0	8±0.3
web-BerkStan	**24*±0.5**	24±0.4	26±0.5	31±0.5	28±0.2	33±0.6	38±0.6	38±0.5	41±0.9
as-skitter	**37*±0.6**	38±0.5	40±0.6	45±0.5	45±0.7	49±0.6	56±0.7	57±0.6	58±0.7
patentcite	**16*±0.3**	18±0.0	19±0.4	20±0.0	22±0.0	23±0.0	28±0.0	30±0.1	28±0.0
in-2004	**41*±0.5**	43±0.5	46±0.4	56±0.3	55±0.6	62±0.5	81±0.4	80±0.5	83±0.4
dblp-author	**10*±0.3**	14±0.0	13±0.0	13±0.0	16±0.0	16±0.0	17±0.1	20±0.0	16±0.0
wb-edu	**90*±0.9**	96±0.4	98±0.2	150±0.5	154±0.5	161±0.3	283±0.3	287±0.3	287±0.4
soc-LiveJournal1	**90*±0.9**	93±0.7	96±0.8	119±0.6	119±0.7	127±1.0	176±0.7	178±1.1	179±0.9
wikipedia-link-de	**317*±1.2**	322±1.5	342±1.2	396±1.2	389±1.6	428±1.2	517±1.7	518±2.0	539±1.6
hollywood-2009	**239*±1.2**	241±0.6	250±0.6	305±0.7	294±1.2	319±0.8	400±1.3	398±1.3	409±1.4
com-orkut	**179*±12.9**	182±15.8	190±12.6	203±15.4	200±17.6	219±16.9	239±19.0	243±21.0	247±16.8
trackers	1508±61.2	1471±54.3	**1452*±16.6**	1858±75.3	1799±69.8	1812±31.3	2503±19.2	2421±79.2	2330±21.4
indochina-2004	**804*±1.2**	810±0.9	817±1.2	1430±1.1	1421±1.2	1464±1.2	2936±1.6	2936±1.5	2945±1.6
uk-2002	**311*±0.7**	329±0.7	330±0.7	547±0.7	556±0.6	572±0.7	1067±0.7	1080±0.6	1070±0.9
arabic-2005	**1291*±3.4**	1313±3.5	1340±3.4	2306±3.5	2296±4.2	2374±3.5	4495±4.0	4503±4.0	4536±3.7
uk-2005	**552*±16.5**	617±19.5	612±25.6	906±21.0	960±23.4	992±25.3	1627±18.3	1690±28.9	1632±18.1
webbase-2001	**1850*±0.7**	1935±0.7	1912±0.9	4014±0.57	4078±0.6	4094±0.8	9069±0.8	9135±0.6	9069±0.9
it-2004	**1926*±10.5**	1985±3.2	1987±11.2	3696±10.1	3687±2.7	3764±14.8	7582±5.8	7599±4.4	7596±14.1

For each of *Ours*, *BC* and *EC*, we can create a variant with SM or VP, leading to 9 program versions whose performance are reported in Table II. In Table II, the three variants of each of *Ours*, *BC* and *EC* constitute a subtable, and the overall best among the 9 versions are highlighted with asterisks.

Surprisingly, our basic GPU algorithm performs the best on all datasets except for *trackers* where VP performs the best. Note that the results are reported based on 100 runs and thus very stable, and VP wins on *trackers* since this dataset has a very biased degree distribution (recall from Table I that the degree std is 2774 while the average degree is 10.2), so the majority of vertices have a low degree and hence their computation cost is low, and the overlapped vertex prefetching effectively hides the memory latency. For all the other datasets, VP is not favorable since computation cost dominates, and the fact that each block has only 31 warps doing the actual computation rather than 32 increases the running time.

SM is not helpful (except on *trackers* where computation cost does not dominate) since, as Fig. 7 shows, we need to run additional instructions for write-location translation (with a case checking), and the *buf*$[i]$-appending operation is very frequent. This overhead outweighs the benefit of having up to $|\mathcal{B}|$ items appended to and read from the shared memory rather than *buf*$[i]$, which could be just a very small fraction of all the k-shell vertices traversed by block i in iteration k.

Both BC and EC slow down the computation as compared with *Ours*. This is because AtomicAdd(.) has been so optimized in the latest GPUs [11] that the contention for result collection is no longer a disadvantage as compared with the old-school compaction approaches [12] that determine result offsets in batch before concurrent writing. After all, compaction runs additional instructions to compute offsets, the cost of which is non-trivial especially given that buffer appending is frequent. Similar observation has been found by CuTS [80] for subgraph matching on GPUs. In addition, BC is often twice as fast as EC (although twice as slow as *Ours*) especially on the last few big datasets. This shows that Ballot scan for warp-level compacting during the scan phase is highly efficient thanks to the use of CUDA primitives __ballot_sync and __popc (c.f. Fig. 8(c)). On the other hand, block-level compacting for large-batch appending backfires, since as Fig. 9 shows, block-level offset computation takes 4 steps where only Warp 0 computes in Stages (2) and (3).

In summary, the Occam's razor principle applies for algorithm design on modern GPU architecture where operations (e.g., AtomicAdd(.)) have been highly optimized in performance, and memory latency is not as high as worthy of hiding by additional thread computations (which can often be more expensive). While our integration of the best optimization practices in the GPU algorithm literature does not help in most cases, our exploration is still valuable in hinting that future works on GPU algorithms should compare with simplistic designs for ablation study to justify the use of optimizations.

Comparison with GPU Baselines. We compare *Ours* with the GPU baselines as shown in Table III, which we have introduced in Sections II and V, where Medusa-Peel, Gunrock and GSwitch all run the edge-centric peeling algorithm while Medusa-MPM runs the vertex-centric MPM algorithm. Medusa-Peel is clearly faster than Medusa-MPM due to less computing workloads, but both of them has graphs that run beyond 1 hour so we force-terminate the programs. The GPU graph-parallel systems also cannot process large graphs and run out of the global memory (OOM) while *Ours* does not have this problem. Overall, Medusa is slower than Gunrock, which is in turn slower than GSwitch. GSwitch is even faster than VETGA that does not use a vertex-centric system, which shows that its execution engine is really well designed and effective. However, Table III shows that GSwitch is still often tens of times slower than *Ours*, showing the benefit of a tailor-made GPU program that achieves native hardware speed.

TABLE III
COMPUTATION TIME OF GPU PROGRAMS (UNIT: MILLISECOND)

Dataset	Ours	VETGA	Medusa-MPM	Medusa-Peel	Gunrock	GSwitch
Amazon0601	**1***	133	34,835	588	38	30
wiki-Talk	**36***	675	> 1hr	12,657	1,236	168
web-Google	**5***	225	44,423	1,986	127	62
web-BerkStan	**24***	1,556	> 1hr	38,141	822	309
as-skitter	**37***	732	> 1hr	22,665	723	238
patentcite	**16***	637	7,964	17,290	1,282	319
in-2004	**41***	2,294	> 1hr	37,089	2,577	523
dblp-author	**10***	122	15,110	2,209	233	78
wb-edu	**90***	10,522	> 1hr	187,674	10,880	5,096
soc-LiveJournal1	**90***	2,868	3,128,970	181,851	7,567	1,343
wikipedia-link-de	**317***	5,674	> 1hr	598,142	54,046	1,240
hollywood-2009	**239***	7,279	> 1hr	455,682	26,442	1,077
com-orkut	**179***	7,186	> 1hr	929,049	10,800	1,635
trackers	**1425***	50,169	> 1hr	> 1hr	208,473	5,072
indochina-2004	**804***	28,350	> 1hr	2,455,986	248,785	10,278
uk-2002	**311***	18,835	> 1hr	1,596,779	44,604	11,070
arabic-2005	**1291***	LD > 1hr	OOM	OOM	OOM	35,190
uk-2005	**552***	LD > 1hr	OOM	OOM	OOM	23,914
webbase-2001	**1850***	LD > 1hr	OOM	OOM	OOM	OOM
it-2004	**1926***	LD > 1hr	OOM	OOM	OOM	OOM

VETGA [22] runs with PyTorch to utilize GPU, and its graph loading code uses a slow NumPy array program which we revise to eliminate NumPy to speed up loading, but even so, it still cannot load the last four big graphs after 1 hour so we force-terminate these VETGA programs. In Table III, "LD $>$ 1hr" means data loading takes more than 1 hour and is thus force-terminated. Table III shows that on graphs that VETGA can load, it is 1 to 2 orders of magnitude slower than *Ours*.

Comparison with CPU Baselines. We compare *Ours* with the CPU baselines as shown in Table IV, which we have introduced in Sections II. Specifically, [51] has implemented the serial BK algorithm, the serial and parallel versions of ParK, PKC (including a slower variant PKC-o [51]) and MPM, so we directly use their implementations [20]. We also include the implementation of BK in the Python library NetworkX [15]. We can see that NetworkX is really not for big graphs as the loading time can go beyond 1 hour, and the running time is many orders of magnitude longer than BZ. Also, the parallel ParK and MPM are often slower than the serial BZ, while PKC can be a few times faster. However, in all cases *Ours* is a clear winner, showing that using a GPU is advantageous even compared with using all 48 CPU cores of a high-end server. In fact, parallel ParK, PKC and MPM are far from achieving 48$\times$ speedup compared with serial variants,

TABLE IV
COMPUTATION TIME OF CPU PROGRAMS (UNIT: MILLISECOND)

Dataset	Ours	NetworkX	BZ	Serial ParK	ParK	Serial PKC-o	PKC-o	MPM	Serial PKC	PKC
amazon0601	**1***	14,889	84	42	102	40	36	79	41	24
wiki-Talk	**36***	163,028	116	434	326	323	143	310	116	93
web-Google	**5***	62,443	150	124	188	112	64	91	103	57
web-BerkStan	**24***	1,393,519	103	213	954	184	113	228	83	87
as-skitter	**37***	301,474	279	270	420	229	132	247	187	110
patentcite	**16***	155,717	878	647	467	587	220	591	519	221
in-2004	**41***	240,000	198	1,053	1,175	884	351	404	186	202
dblp-author	**10***	136,541	674	334	183	316	142	388	294	133
wb-edu	**90***	320,773	1,868	6,658	3,776	6,630	1,637	4,530	1,011	470
soc-LiveJournal1	**90***	520,355	2,358	2,919	2,066	2,389	891	2,663	1,336	678
wikipedia-link-de	**317***	LD > 1hr	2,432	5,206	2,722	4,647	1,573	2,596	1,724	964
hollywood-2009	**239***	1,630,000	1,208	2,151	2,177	1,896	916	1,711	763	663
com-orkut	**179***	1,814,440	4,456	2,256	4,431	203	2,700	21,533	2,156	2,781
trackers	**1425***	LD > 1hr	5,295	11,389	4,772	11,304	4,124	15,869	4,426	3,081
indochina-2004	**804***	LD > 1hr	2,378	63,581	18,173	64,137	14,454	6,434	2,991	2,636
uk-2002	**311***	LD > 1hr	8,782	26,943	9,390	26,834	6,741	5,441	3,655	1,618
arabic-2005	**1291***	LD > 1hr	14,733	106,899	29,903	106,306	25,857	14,304	8,239	4,176
uk-2005	**552***	LD > 1hr	24,941	37,212	13,034	36,976	9,720	8,821	8,910	3,917
webbase-2001	**1850***	LD > 1hr	38,590	189,330	45,044	189,208	39,342	48,452	14,834	5,474
it-2004	**1926***	LD > 1hr	31,732	195,887	54,286	195,609	45,637	39,346	15,171	7,104

due to memory latency and contention for buffer appending.

Peak GPU Memory Usage. We also let a daemon program continuously run the "nvidia-smi" command to log the GPU memory consumption when running experiment of Tables II and III. Table V reports the peak global memory usage results, where our peeling algorithm is clearly the overall winner, while Gunrock and GSwitch win on some small datasets.

Case Study. Having a lightening fast k-core decomposition program like *Ours* is important when studying networks that change dynamically so that k-core decomposition can be performed frequently or even continuously on the network snapshots, such as studying the gene co-expression and protein-protein interaction networks during the onset of Arabidopsis leaf senescence [62], and tracking an evolving interaction network such as online social networks or collaboration networks. Here, we conduct a case study using the citation network from [7] which consists of papers chosen from Arnet-Miner [73], [74] that fall in 10 topics such as Data Mining and Database Systems, as well as their citation relationship. Each paper is associated with the year of publishing as well as the list of authors. We preprocess the paper citation network into an author interaction network, where an edge (u, v) is added if there exists a paper (co-)authored by u that cited another paper (co-)authored by v, and vice versa.

We consider two co-citation networks: G_1 (resp. G_2) which includes all papers in or before 1995 (resp. 2000), whose $k_{max} = 12$ (resp. 18) and whose k_{max}-core denoted by S_1 (resp. S_2) has 81 (resp. 107) authors. In Figure 10, we use

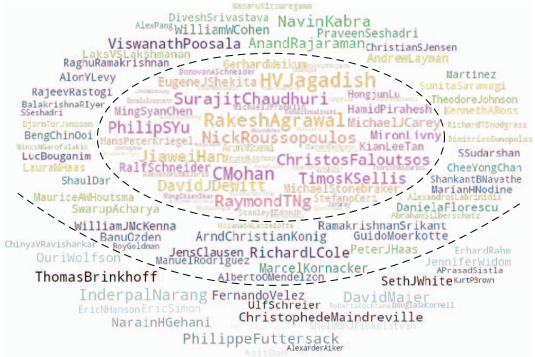

Fig. 10. Case Study: Co-Citation Network Analysis

word cloud to show (1) $S_1 \cap S_2$ in the center with authors like PhilipSYu and HVJagadish who were most active in both G_1 and G_2, (2) $(S_2 - S_1)$ in the middle ring with authors like ChristianSJensen and BengChinOoi who became most active in G_2, and (3) $(S_1 - S_2)$ in the bottom with authors fall out of the most active k_{max}-core moving from 1995 to 2000.

VII. CONCLUSION AND FUTURE WORK

We have designed a highly-efficient peeling algorithm for k-core decomposition on a GPU, which achieves native speed compared with other GPU solutions built with PyTorch or vertex-centric graph-parallel systems. Our GPU algorithm is also faster than existing serial and parallel CPU solutions.

TABLE V
PEAK GLOBAL MEMORY USAGE (UNIT: MB)

Dataset	Ours	SM	VP	EC	BC	VETGA	Medusa-MPM	Medusa-Peel	Gunrock	GSwitch
amazon0601	303	305	305	409	409	1,025	427	429	325	**253***
wiki-Talk	751	751	751	893	893	1,291	N/A	579	**511***	663
web-Google	737	737	737	845	845	1,077	521	527	**467***	611
web-BerkStan	755	755	755	897	897	1,109	N/A	635	**559***	647
as-skitter	797	797	797	939	939	1,239	N/A	861	**745***	747
patentcite	**855***	**855***	**855***	997	997	1,449	1,137	1,161	991	897
in-2004	811	811	811	953	953	1,253	N/A	959	829	**767***
dblp-author	833	833	833	881	881	1,595	959	993	855	**531***
wb-edu	**1,125***	**1,125***	**1,125***	1,267	1,267	2,313	N/A	2,645	2,223	1,529
soc-LiveJournal1	**1,063***	**1,063***	**1,063***	1,205	1,205	1,949	2,353	2,387	2,015	1,329
wikipedia-link-de	**1,315***	**1,315***	**1,315***	1,457	1,457	2,403	N/A	3,935	3,307	1,809
hollywood-2009	**1,139***	**1,139***	**1,139***	1,281	1,281	2,051	N/A	2,927	5,097	1,423
com-orkut	**1,617***	**1,617***	**1,617***	1,720	1,720	2,925	N/A	5,751	4,815	2,401
trackers	**1,981***	**1,981***	**1,981***	2,123	2,123	4,603	N/A	N/A	6,131	3,519
indochina-2004	**1,907***	**1,907***	**1,907***	2,049	2,049	4,817	N/A	7,391	8,851	3,057
uk-2002	**2,837***	**2,837***	**2,837***	2,979	2,979	5,589	N/A	12,707	10,597	5,095
arabic-2005	**5,097***	**5,097***	**5,097***	5,239	5,239	N/A	N/A	N/A	N/A	9,679
uk-2005	**5,811***	**5,811***	**5,811***	5,953	5,953	N/A	N/A	N/A	N/A	13,697
webbase-2001	**6,319***	**6,319***	**6,319***	6,461	6,461	N/A	N/A	N/A	N/A	N/A
it-2004	**8,851***	**8,851***	**8,851***	8,993	8,993	N/A	N/A	N/A	N/A	N/A

Currently, our GPU program will fail if *buf*$[i]$ of any block overflows, which poses a limit on the graph size that a GPU can process. As a future work, we plan to extend our algorithm for multi-GPU computation, where as in [44], we can partition a graph among worker GPUs running our kernels, but degree updates of border vertices would be aggregated afterwards, which can be computed at a master GPU. Moreover, the updates may cause new border vertices to be in k-shell, so more than one round may be needed to compute a k-shell.

REFERENCES

[1] 2002 web crawl of .uk domain. https://sparse.tamu.edu/LAW/uk-2002.

[2] 2004 Indochina web crawl. https://sparse.tamu.edu/LAW/indochina-2004.

[3] 2004 web crawl of .it domain. https://sparse.tamu.edu/LAW/it-2004.

[4] 2005 web crawl of Arabic domains. https://sparse.tamu.edu/LAW/arabic-2005.

[5] 2005 web crawl of .uk domain. https://sparse.tamu.edu/LAW/uk-2005.

[6] Amazon product co-purchasing network from June 1 2003. https://snap.stanford.edu/data/amazon0601.html.

[7] ArnetMiner Citation Network. https://lfs.aminer.cn/lab-datasets/soinf/.

[8] Citation network among US Patents. https://snap.stanford.edu/data/cit-Patents.html.

[9] DBLP collaboration network. http://konect.cc/networks/dblp-author/.

[10] *.edu web pages, A(i,j)=1 if page i links to page j (2001). https://sparse.tamu.edu/Gleich/wb-edu.

[11] GPU Atomic Memory Operations. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#atomic-ops.

[12] GPU Compact Operation. https://www.youtube.com/watch?v=GyYfg3ywONQ&list=PLnH7E0IG44jFfiQBd_Ov7FmYHq8SZx6r0&index=170.

[13] Hollywood movie actor network. https://law.di.unimi.it/webdata/hollywood-2011/.

[14] Internet topology graph, from traceroutes run daily in 2005. https://snap.stanford.edu/data/as-Skitter.html.

[15] k-core decomposition in NetworkX. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.core.core_number.html#networkx.algorithms.core.core_number.

[16] Large web crawl in 2001. https://sparse.tamu.edu/LAW/webbase-2001.

[17] LiveJournal online social network. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[18] Network of internet domains and the trackers they contain, such as the Facebook button and Google Analytics. http://konect.cc/networks/trackers-trackers/.

[19] Orkut online social network. https://snap.stanford.edu/data/com-Orkut.html.

[20] PKC. https://github.com/humayunk1/PKC.git.

[21] Small web crawl of .in domain. https://sparse.tamu.edu/LAW/in-2004.

[22] VETGA. https://github.com/mexuaz/vetga.git.

[23] Web graph from Google. https://snap.stanford.edu/data/web-Google.html.

[24] Web graph of Berkeley and Stanford. https://snap.stanford.edu/data/web-BerkStan.html.

[25] Wikilinks of the Wikipedia in the German language (de). http://konect.cc/networks/wikipedia_link_de/.

[26] Wikipedia talk (communication) network. https://snap.stanford.edu/data/wiki-Talk.html.

[27] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *Networks Heterog. Media*, 3(2):371–393, 2008.

[28] G. D. Bader and C. W. Hogue. Analyzing yeast protein–protein interaction data obtained from different sources. *Nature biotechnology*, 20(10):991–997, 2002.

[29] D. Bakunas-Milanowski, V. Rego, J. Sang, and C. Yu. Efficient algorithms for stream compaction on gpus. *Int. J. Netw. Comput.*, 7(2):208–226, 2017.

[30] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[31] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *PPoPP*, pages 235–248. ACM, 2017.

[32] E. Blelloch Guy. Prefix sums and their applications. Technical report, Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, 1990.

[33] F. Bonchi, A. Khan, and L. Severini. Distance-generalized core decomposition. In *SIGMOD*, pages 1006–1023. ACM, 2019.

[34] 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*, 104(27):11150–11154, 2007.

[35] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62. IEEE Computer Society, 2011.

[36] Y. Cheng, C. Lu, and N. Wang. Local k-core clustering for gene networks. In *IEEE BIBM*, pages 9–15. IEEE Computer Society, 2013.

[37] D. Chu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Y. Xia, and C. Zhang. Finding the best k in core decomposition: A time and space optimal solution. In *ICDE*, pages 685–696. IEEE, 2020.

[38] D. Chu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang. Hierarchical core decomposition in parallel: From construction to subgraph search. In *ICDE*, pages 1138–1151. IEEE, 2022.

[39] A. Conte, D. Firmani, C. Mordente, M. Patrignani, and R. Torlone. Fast enumeration of large k-plexes. In *KDD*, pages 115–124. ACM, 2017.

[40] Q. Dai, R. Li, L. Qin, G. Wang, W. Yang, Z. Zhang, and Y. Yuan. Scaling up distance-generalized core decomposition. In *CIKM*, pages 312–321. ACM, 2021.

[41] N. S. Dasari, D. Ranjan, and M. Zubair. Park: An efficient algorithm for k-core decomposition on multicore processors. In *IEEE BigData*, pages 9–16. IEEE Computer Society, 2014.

[42] R. Dathathri, G. Gill, L. Hoang, H. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, pages 752–768. ACM, 2018.

[43] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In O. Cheong, K. Chwa, and K. Park, editors, *ISAAC, Part I*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2010.

[44] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *SIGMOD Conference*, pages 495–510. ACM, 2017.

[45] Z. Fu, B. B. Thompson, and M. Personick. Mapgraph: A high level API for fast development of high performance graph analytics on gpus. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 2:1–2:6. CWI/ACM, 2014.

[46] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *ICDM*, pages 201–210. IEEE Computer Society, 2011.

[47] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowl. Inf. Syst.*, 35(2):311–343, 2013.

[48] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.

[49] G. Guo, D. Yan, L. Yuan, J. Khalil, C. Long, Z. Jiang, and Y. Zhou. Maximal directed quasi -clique mining. In *ICDE*, pages 1900–1913. IEEE, 2022.

[50] W. D. Hillis and G. L. S. Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[51] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *IPDPS Workshops*, pages 1482–1491. IEEE Computer Society, 2017.

[52] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *VLDB J.*, 31(4):649–674, 2022.

[53] W. Khaouid, M. Barsky, S. Venkatesh, and A. Thomo. K-core decomposition of large networks on a single PC. *Proc. VLDB Endow.*, 9(1):13–23, 2015.

[54] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC*, pages 239–252. ACM, 2014.

[55] 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*, 6(11):888–893, 2010.

[56] X. Liao, Q. Liu, J. Jiang, X. Huang, J. Xu, and B. Choi. Distributed d-core decomposition over large directed graphs. *Proc. VLDB Endow.*, 15(8):1546–1558, 2022.

[57] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[58] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.

[59] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[60] A. Mehrafsa, S. Chester, and A. Thomo. Vectorising k-core decomposition for GPU acceleration. In E. Pourabbas, D. Sacharidis, K. Stockinger, and T. Vergoulis, editors, *SSDBM*, pages 24:1–24:4. ACM, 2020.

[61] K. Meng, J. Li, G. Tan, and N. Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *PPoPP*, pages 201–213. ACM, 2019.

[62] B. Mishra, Y. Sun, T. Howton, N. Kumar, and M. S. Mukhtar. Dynamic modeling of transcriptional gene regulatory network uncovers distinct pathways during the onset of arabidopsis leaf senescence. *NPJ systems biology and applications*, 4(1):1–4, 2018.

[63] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. In C. Gavoille and P. Fraigniaud, editors, *PODC*, pages 207–208. ACM, 2011.

[64] T. NVIDIA. Nvidia® tesla® p100–the most advanced data center accelerator ever built. Technical report, Technical Report WP-08019-001, 2017.

[65] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community detection in social media - performance and application considerations. *Data Min. Knowl. Discov.*, 24(3):515–554, 2012.

[66] M. Pellegrini, F. Geraci, and M. Baglioni. Detecting dense communities in large social and information networks with the core & peel algorithm. *CoRR*, abs/1210.3266, 2012.

[67] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary. A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components. *CoRR*, abs/1302.6256, 2013.

[68] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, 2013.

[69] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.*, 25(3):425–447, 2016.

[70] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[71] S. Sengupta, M. J. Harris, M. Garland, and J. D. Owens. *Efficient parallel scan algorithms for many-core gpus*. eScholarship, University of California, 2011.

[72] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Trans. Knowl. Data Eng.*, 30(1):29–42, 2018.

[73] J. Tang, J. Sun, C. Wang, and Z. Yang. Social influence analysis in large-scale networks. In *KDD*, pages 807–816. ACM, 2009.

[74] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, pages 990–998. ACM, 2008.

[75] A. Tripathy, F. Hohman, D. H. Chau, and O. Green. Scalable k-core decomposition for static graphs using a dynamic graph data structure. In *IEEE BigData*, pages 1134–1141. IEEE, 2018.

[76] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *PPoPP*, pages 38–52. ACM, 2019.

[77] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*, pages 11:1–11:12. ACM, 2016.

[78] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition at web scale. In *ICDE*, pages 133–144. IEEE Computer Society, 2016.

[79] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Trans. Parallel Distributed Syst.*, 33(1):129–143, 2022.

[80] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam. cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In *SC*. ACM, 2021.

[81] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Found. Trends Databases*, 7(1-2):1–195, 2017.

[82] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, 2014.

[83] K. Yao and L. Chang. Efficient size-bounded community search over large networks. *Proc. VLDB Endow.*, 14(8):1441–1453, 2021.

[84] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin. When engagement meets similarity: Efficient (k, r)-core computation on social networks. *Proc. VLDB Endow.*, 10(10):998–1009, 2017.

[85] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1543–1552, 2014.