Geld: Load-balanced D-Core Decomposition for Consumer GPUs

CHENG HUANG, Aarhus University, Denmark
JOHANNES LANGGUTH, Simula Research Laboratory, Norway
XING CAI, University of Oslo, Norway
DAVIDE MOTTIN, Aarhus University, Denmark
IRA ASSENT, Aarhus University, Denmark

D-core decomposition (DCD) identifies groups of nodes with strong cohesion at different levels of granularity, supporting e.g. clustering and community detection in *directed graphs*. Given the inefficiency of computing DCD, the computational power of GPUs is an attractive target. However, existing algorithms are ill-suited due to their restrictive programming model, as well as memory constraints in the widely available consumer-level GPUs. In particular, we show that any trivial adaptation of peeling algorithms, state-of-the-art for DCD, suffers from workload imbalance, redundant operations, and repeated atomic operations. To address these challenges, we propose a novel dynamic strategy to balance the computation on nodes with varying vertex degree and an enumeration-based lightweight memory footprint with greatly reduced thread contention. We conduct extensive experiments on 12 datasets showing that `Geld` achieves an average speedup of 31× over GPU peeling DCD, 6× over GPU H-Index DCD, 22× over the best existing parallel DCD algorithm with 33% memory efficiency, and 3 orders of magnitude over the best serial DCD algorithm.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Parallel algorithms**.

Additional Key Words and Phrases: D-core Decomposition, GPU, Parallel Algorithms, Graph Algorithms

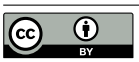
ACM Reference Format:
Cheng Huang, Johannes Langguth, Xing Cai, Davide Mottin, and Ira Assent. 2026. `Geld`: Load-balanced D-Core Decomposition for Consumer GPUs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 187 (June 2026), 26 pages. https://doi.org/10.1145/3802064

1 Introduction

Connections among individuals, entities, or organizations naturally follow a direction. For instance, users in a social network follow one another [35], customers exchange money in transactions [11], or web pages link to other pages. To model such directed interactions, *directed graphs* provide a convenient abstraction.

A notable family of techniques for analyzing connectivity in such graphs is Cohesive Subgraph Mining (CSM) [2, 7, 12, 43]. CSM identifies densely connected subgraphs, e.g., communities, that satisfy specific cohesiveness constraints. The simplest and most popular cohesive subgraph mining model for undirected graphs is k-core [36], which is the largest subgraph where each vertex has at least k neighbors. *D-core (also (k, l)-core)* [15] extends k-core to directed graphs by discovering the largest subgraph where each vertex has at least k in-neighbors and l out-neighbors.

Authors' Contact Information: Cheng Huang, Aarhus University, Aarhus, Denmark, cheng@cs.au.dk; Johannes Langguth, Simula Research Laboratory, Oslo, Norway, langguth@simula.no; Xing Cai, University of Oslo, Oslo, Norway, xingca@ifi.uio.no; Davide Mottin, Aarhus University, Aarhus, Denmark, davide@cs.au.dk; Ira Assent, Aarhus University, Aarhus, Denmark, ira@cs.au.dk.



ACM 2836-6573/2026/6-ART187
https://doi.org/10.1145/3802064

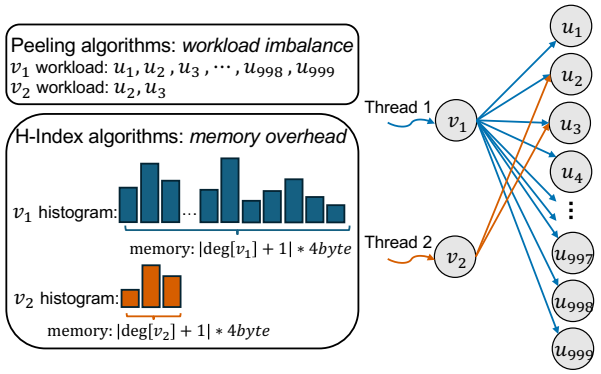


Fig. 1. Peeling-based vs. H-Index-based algorithms.

Applications. D-core is widely adopted in key real-world scenarios. In community search [8, 13], D-core provides an effective model for studying groups in directed graphs, where the joint in- and out-degree constraints enable interpretability of groups' relationships. In collaboration networks [15], D-core unveils collaboration patterns, such as paper mills, where the direction of citations also indicates potential temporal patterns. In social networks, D-core is useful for analyzing popularity, reputation, and influence propagation [14]: users belonging to D-cores with high in-degree are more likely to be followed by other influential users within the same D-core. These applications often involve large-scale graphs, making efficiency a first-class citizen in any practical scenario.

D-core decomposition (DCD) aims to compute **all** non-empty D-cores of a directed graph for every pair of k and l, finding communities at different granularity levels. DCD's utility across application domains and its inherent computational demands have sparked interest in its acceleration, with two families of algorithms: *peeling-based* [13, 15, 26] and *H-Index-based* [22, 26]. Peeling algorithms [13, 15, 26] iteratively remove vertices that do not satisfy the (k,l)-constraint while updating the degrees of their neighbors. They have both sequential and multi-core CPU adaptations; however, they are not tailored for the computational model of the GPU. To properly investigate the shortcomings of peeling, we propose an adaptation of the peeling algorithm `GPeel` for GPUs. Our investigation concludes that the peeling algorithm with vertex-to-vertex dependency may cause severe thread workload imbalance issues. Moreover, as multiple vertices share the same neighbors, updating the common neighbors' degree results in heavy thread contention, necessitating atomic operations that greatly reduce efficiency.

Observation 1 (Workload imbalance). *In Fig. 1, thread* 1 *and thread* 2 *update in parallel their neighbor degrees. However, since v_1 has 999 neighbors while v_2 has only 2, this results in a heavy imbalance workload, meaning thread 2 risks idling until thread 1 is done.*

In H-index-based algorithms [22, 26], each vertex performs local updates to estimate its core number, an upper bound on its out-degree. This requires maintaining an auxiliary histogram whose size can reach the vertex's out-degree. However, the widely available and by far most affordable consumer GPUs typically provide up to 24 GB of memory, while vertex degrees in real-world graphs can reach millions, limiting the practicality of H-index methods. Moreover, similar to peeling, such algorithms also suffer from workload imbalance issues, as each vertex must iterate over its neighbors to update the histogram for estimating the core number.

Observation 2 (Memory overhead). *In Fig. 1, both v_1 and v_2 maintain an auxiliary per-vertex histogram of size out degree + 1, as a vertex's core number is upper bounded by its out degree.*

To address these issues, we introduce the first GPU-accelerated load-balanced algorithm for DCD (Geld). Geld targets *consumer GPUs*, a *more accessible, affordable, widely available, and privacy-friendly* alternative to data-centric GPUs. Due to their affordability [23], consumer GPUs are pervasive in research labs and universities as the prime choice for large-scale data analytics [9, 39]. Moreover, they can typically be obtained and deployed immediately, whereas high-end data-center GPUs often have lead times of several months [40]. In addition, consumer GPUs also run on local desktops, so sensitive data does not leave the machine, where data-center GPUs operate in shared cluster environments [33, 34], giving rise to concerns in highly private domains in medicine, biology, etc.

However, this design also introduces new challenges, as consumer GPUs are more sensitive to workload imbalance and atomic contention between threads compared with enterprise GPUs [31]. These issues are particularly critical for graph algorithms, where highly irregular degree distributions and frequent atomic updates can severely degrade parallel efficiency. To tackle these challenges, we develop two key design strategies. Firstly, a degree-aware strategy that adaptively assigns vertices with different degrees to achieve workload balance. Second, an enumeration-based strategy that is both memory-efficient and atomic-free, eliminating the need for auxiliary structures. Moreover, we introduce three pruning techniques to further reduce the computation space and enhance overall efficiency. In summary, we contribute with:

- The first study of D-core decomposition on large directed graphs using consumer GPUs.
- A comprehensive analysis of GPU-based peeling for DCD reveals three key bottlenecks: high atomic contention, workload imbalance, and redundant scans (Section 5).
- A GPU-native algorithm combining degree-aware and enumeration-based strategies for balanced workloads, memory efficiency, and high performance (Section 6).
- Extensive experiments on 12 real-world datasets show that Geld achieves an average of 31× speedup over the best GPU peeling DCD algorithm, 6.02× speedup over the best GPU H-Index DCD algorithm with 33% better memory efficiency, 21× speedup over the best parallel CPU algorithm, and over three orders of magnitude speedup compared to the best serial baseline (Section 7).

2 Related Work

We review the algorithms for k-core decomposition and provide an overview of existing approaches to D-core decomposition.

k-core decomposition on CPUs and distributed systems. k-core decomposition aims to identify a maximal subgraph where each vertex has a degree of at least k, and it has been extensively studied in various contexts [2, 27, 37, 38, 41]. Peeling algorithms remove the vertex with the smallest degree from the graph until no vertex remains. BZ [3] is the first peeling algorithm with linear complexity in the number of edges. ParK [10] extends BZ to multi-core CPUs: It scans the vertex set in parallel and peels vertices with degree at most k into a global buffer. Each thread extracts a vertex from this buffer, updates the degree of its neighbors in parallel, and adds those neighbors to the buffer if their degree is at most k. PKC [18] improves upon ParK by reducing synchronization overhead. Instead of relying on a global buffer, the threads in PKC maintain their own buffer, enhancing efficiency. MDM [29] addresses k-core decomposition in distributed settings using the h-index concept [16]: a vertex belongs to the k-core if it has at least k neighbors also in the k-core. Each vertex iteratively refines its core number based on its neighbors' core numbers until convergence.

k-core decomposition on GPUs. Motivated by the prevalence of GPUs, several works have explored accelerating k-core decomposition on GPUs to achieve higher efficiency. Vectr [28] reformulates the peeling algorithm as vector operations and leverages optimized primitives in

PyTorch for efficient SIMD execution. PP-dyn [1] optimizes the peeling algorithm with highly optimized CUDA kernels. It gathers the peeled vertices into multiple block-level frontiers and assigns a warp of threads to update their neighbors' degrees. Since multiple peeled vertices may connect to the same neighbor, atomic operations are unavoidable. However, the increased use of atomic operations raises efficiency concerns. To mitigate this, PeelOne [42] introduces a new atomic primitive to prevent a vertex's degree from dropping below k after an atomic decrement if it is in the k-core, but atomic still remains a bottleneck. Apart from peeling, HistoCore [42] applies the h-index method by maintaining an auxiliary histogram for each vertex to compute the core number. However, the histogram maintenance incurs synchronization overhead and substantial memory consumption, as each histogram scales with the vertex's degree. Moreover, most real-world graphs follow the power-law distribution, causing an imbalanced workload among GPU threads, which is harmful for the performance. However, none of the existing GPU algorithms [1, 28, 42] address this issue.

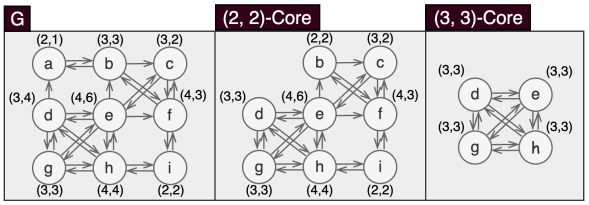
D-core decomposition on CPUs and distributed systems. Vertices in directed graphs have both in- and out-neighbors. In such graphs, DCD [15] aims to compute the largest subgraphs where each vertex satisfies a minimum in-degree k and out-degree l. For each k and l combination, Trim [15] iteratively removes vertex with in-degree $< k$ and out-degree $< l$ until the remaining vertices satisfy the (k, l)-constraint. `Peeling` [13] reduces the search space by computing a k-list for each value of k and iteratively peeling vertices with the smallest out-degree. `ParPeel` [26] parallelizes `Peeling` on a multi-core CPU by peeling vertices with the smallest out-degree in parallel and adding them to each thread's local buffer. Then, each thread retrieves a vertex from its own buffer and updates the in- and out-degrees of its neighbors. While efficient, `ParPeel` still suffers from scalability issues on billion-edge or denser graphs due to severe thread contention and workload imbalance. Distributed solutions [22] using the h-index property of D-core to compute *out-core numbers*, where the out-core number of a vertex v is the largest out-degree l such that v belongs to a (k, l)-core. Each vertex refines its out-core number using its neighbors' out-core numbers until convergence. `Shell` [26] follows the same h-index approach on a multi-core CPU with per-vertex histograms similar to HistoCore [42], which again introduces memory overhead and workload imbalance, limiting scalability on CPU threads.

Prior research studies k-core decomposition on CPUs [3, 10, 18, 25], distributed systems [29] and GPUs [1, 28, 42]. Yet, none of the existing GPU algorithms for k-core decomposition apply to DCD, as D-core is more complex by virtue of both in- and out-degree constraints. Although there exist parallel DCD algorithms for CPUs [13, 15, 26] and distributed systems [22], DCD on GPUs remains unexplored. In general, graph algorithms on GPUs face inherent challenges due to workload imbalance and memory overhead, an open problem we tackle for DCD in this paper.

3 Problem Definition

In this section, we formalize the problem of D-core decomposition and review its core properties. A *directed graph* is a pair $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ edges. An edge $(u, v) \in E$ is a directed connection from vertex u to vertex v. The in-neighbors of a vertex $v \in V$ are $N^+(v) = \{u|(u, v) \in E\}$; similarly, the out-neighbors $N^-(v) = \{u|(v, u) \in E\}$. The in-degree is $\deg^+[v] = |N^+(v)|$ and out-degree $\deg^-[v] = |N^-(v)|$. A graph $H = (V_H, E_H)$ is a subgraph of G iff $V_H \subseteq V$ and $E_H \subseteq E \cap (V \times V)$. Given a subgraph $H = (V_H, E_H)$ of G, the in-neighbors of a vertex $v \in V_H$ in H are denoted $N^+_H(v)$ with in-degree $\deg^+_H[v] = |N^+_H(v)|$ and out-neighbors $N^-_H(v)$ with out-degree $\deg^-_H[v] = |N^-_H(v)|$.

Definition 1 (D-core [15])**.** *Given a directed graph $G = (V, E)$ and two numbers $k, l \in \mathbb{N}$, a D-core of G, is a subgraph $H = (V_H, E_H)$ such that (a) for each vertex $v \in H$, its in-degree $\deg^+_H[v] \geq k$ and*



K \ V	a	b	c	d	e	f	g	h	i
0	1	2	2	3	3	2	3	3	2
1	1	2	2	3	3	2	3	3	2
2	1	2	2	3	3	2	3	3	2
3	-	-	-	3	3	-	3	3	-

Fig. 2. D-Core examples (graphs in top row) and out-core numbers under in-degree constraint k (table in bottom row)

out-degree v $\deg^-_H[v] \geq l$, *and (b)* H *is maximal, i.e., there is no larger* $V'_H \subseteq V$, $V_H \subseteq V'_H$ *for which (a) holds.* H *is a* (k,l)*-core, denoted* C^l_k.

Example 1. *Figure 2 (Top) shows a directed graph* G *with 9 nodes, along with its* $(2,2)$*-core and* $(3,3)$*-core.* G *also satisfies the* $(2,1)$*-core condition, as each vertex has in-degree* ≥ 2 *and out-degree* ≥ 1.
D-cores obey *uniqueness* and *hierarchical structure* properties [15]:
Property 1 (Uniqueness)**.** *Given a directed graph* $G = (V, E)$ *and thresholds* k *and* l*, there is only one* (k, l)*-core.*

Property 2 (Hierarchical Structure)**.** *Given directed graph* $G = (V, E)$*, thresholds* k, l*, the* $(k, l{+}1)$*-core and the* $(k+1, l)$*-core of* G *are subgraphs of the* (k, l)*-core.*

Example 2. *Figure 2: lowering in-degree constraint* k *or out-degree constraint* l *means growing D-cores. E.g., the* $(2,2)$*-core is a subgraph of the* $(2,1)$*-core, and* $(3,3)$*-core is a subgraph of the* $(2,2)$*-core.*

The **out-core number** characterizes the vertices of a D-core [22]:

Definition 2 (Out-Core Number)**.** *Given directed graph* $G = (V, E)$*, number* k*, the out-core number of vertex* v *on* k*,* $C_k[v]$*, is the largest out-degree* l *such that* v *is in* (k, l)*-core:* $C_k[v] = \max\{l \mid v \in C^l_k\}$.

Example 3. *Figure 2: for in-degree constraint* $k = 2$*, vertex a belongs at most to the* $(2,1)$*-core:* $C_2[a] = 1$*; b, c, f, i at most the (2, 2)-core:* $C_2[b] = C_2[c] = C_2[f] = C_2[i] = 2$*; d, e, g, h at most the (2, 3)-core:* $C_2[d] = C_2[e] = C_2[g] = C_2[h] = 3$.

D-core Decomposition aims to compute **all** D-cores of a directed graph G for $k \geq 0$ and $l \geq 0$. It is **equivalent** to determining the out-core number for each vertex v in G given k [22]:

Problem 1 (D-core Decomposition)**.** *Given directed graph* $G = (V, E)$*, D-core decomposition computes the out-core number* $C_k[v]$*, for each vertex* $v \in V$*, for all* $0 \leq k \leq k_{\max}$*, where* $k_{\max}$ *is the maximum in-degree constraint such that* $(k, 0)$*-core is non-empty.*

4 Algorithmic challenges

We propose to exploit GPU parallelism for efficient D-core decomposition, but the GPU computational model incurs major algorithmic design challenges. GPUs offer a scalable architecture for parallel computation in a SIMT *(Single Instruction Multiple Thread)* model. Each GPU contains multiple SMs (*Streaming Multiprocessors*), and each SM schedules and executes multiple warps concurrently. A *warp* consists of 32 threads that share the same program counter; thus, all threads

in a warp execute the same instruction simultaneously. Therefore, *thread divergence*, where threads in a warp follow different execution paths, may severely degrade performance.

Challenge C1. *How can we design a GPU-friendly D-core algorithm reducing thread divergence from varying vertex degrees?*

In Nvidia's GPU access framework CUDA, the CPU invokes a special function known as *kernel*, which is executed directly on GPUs. Threads are organized hierarchically, typically in *blocks* and then *grids*. In CUDA, a grid organizes thread blocks in one to three dimensions and assigns each block a unique index, starting from 0 in each dimension. Each block assigns a unique thread index to every thread in each dimension, also starting from 0. Each thread has a global thread index, which we denote as tid. Only threads within the same block can synchronize at specific points during execution. This imposes a constraint on algorithm design, as it requires computation in different blocks to be independent.

GPU memory is organized into levels: Each thread has its own restricted *register memory*, threads in the same block share their *shared memory*. All threads share *global memory*. Threads access register and shared memory faster than global memory; thus, it is paramount that frequently accessed data resides in shared memory to improve the throughput of the algorithm. Yet, when multiple threads access the same memory concurrently, *race conditions* may occur: uncoordinated writes corrupt results. Atomic operations avoid such errors, but reduce concurrency and thus efficiency.

Challenge C2. *How can we design an efficient D-core decomposition algorithm on the GPU given limited fast memory?*

GPUs are optimized for *coalesced memory access*, by which adjacent threads access consecutive memory addresses to maximize memory throughput. However, in graph algorithms, irregular memory access patterns and significant computational imbalance often arise due to power-law degree distributions and skewed vertex degrees. These irregularities not only degrade memory performance but also lead to severe load imbalance across threads and SMs.

Challenge C3. *How can we load-balance between threads for D-core decomposition on GPU to handle skewed graph structures?*

Next, we present our algorithmic solutions that address these challenges. We analyze the complexity of our algorithms using the well-established *work-span model* [17]: the ***work*** of an algorithm refers to the total number of operations executed, the ***span*** denotes the length of the longest chain of dependent operations.

5 Problem Analysis

We start with a natural question: *can peeling algorithms be readily adapted to solve DCD efficiently on GPUs?* Peeling is a popular approach for cohesive subgraph mining due to its conceptual simplicity and inherent parallelism for multi-core CPUs [10, 18, 24–26] and GPUs [1, 28, 42]. However, given the bi-directional computation in DCD, which is more complex than e.g. uni-directional k-core decomposition, the answer is far from trivial. We therefore adapt peeling to DCD on the GPU for an in-depth analysis.

Algorithm 1 outlines `GPeel`, our peeling-based DCD adaptation for GPUs. In a nutshell, for a given in-degree constraint k, `GPeel` incrementally increases out-degree constraint l, thereby continuously

peeling vertices until all are removed. GPeel follows *block-level frontier-based* design [1, 26, 42], where vertices satisfying the same degree constraints form frontiers $\mathcal{F}$ and peel in parallel. For each in-degree k, we initialize $\mathcal{P}$, to track the peel status of each vertex, where $\mathcal{P}[v] = 0$ indicates that vertex v has not been peeled (Line 2). For each k and l (Lines 5-10), we collect unpeeled vertices with in-degree less than k or out-degree equal to l, and assign them into multiple block-level frontiers [1, 42], which distribute the peeling workload across thread blocks to fully exploit GPU parallelism (Line 6). This multi-frontier design also reduces atomic contention compared with maintaining a single large frontier. Within each frontier, we do not rely on the degree array $\deg^-$ to represent out-core numbers, as done in previous work [1, 26]. Instead, we maintain and update vertices using $C_k[v] = l$, indicating that, for a given k, vertex v belongs to at most the (k, l)-core. This design avoids the additional atomic updates required to restore a vertex's degree when it temporarily falls below l during peeling, thereby reducing the number of atomic operations by half. We then mark $\mathcal{P}[v] = 1$, indicating that we peel the vertex from subsequent updates.

Algorithm 1: GPeel

```
   Data: G = (V, E)
   Result: C: The coreness for each vertex
1  for k ← 0 to k_max do
2      P[·] ← 0_(v∈V)
3      l ← 0
4      while ∃P[v] = 0 do
5          A ← {v | v ∈ V, P[v] = 0, (d+[v] < k ∨ d−[v] = l)}
6          {F_i}_{i=0}^{p} ← Distribute(A)
7          while ∃F_i ≠ ∅ do                                   // In parallel
8              forall v ∈ F_i do                               // In parallel
9                  C_k[v] ← l, P[v] ← 1
10             F_i ← Peel(F_i, P, k, l)
11         l ← l + 1
12 return C
13 Function Peel(F_i, P, k, l):
14     foreach v ∈ F_i do                                      // In parallel
15         foreach u ∈ N+(v) do                                // In parallel
16             δ ← atomic_sub(deg−[u], 1)
17             if δ = l + 1 ∧ atomic_cas(P[u], 0, 1) = 0 then
18                 Add u to F_i, C_k[u] ← l
19         foreach u ∈ N−(v) do                                // In parallel
20             δ ← atomic_sub(deg+[u], 1)
21             if δ = k ∧ atomic_cas(P[u], 0, 1) = 0 then
22                 Add u to F_i, C_k[u] ← l
```

To peel each vertex v in the frontier, we update the in-degree of its unpeeled out-neighbors and out-degree of unpeeled in-neighbors. After degree adjustment, if vertex u belongs to at most the (k, l)-core, we add it to the corresponding frontier and set its core number $C_k[u] = l$. As multiple threads update a vertex u's in- and out-degrees concurrently, each update must be atomic (Lines 16, 19). Also, a vertex whose neighbors are updated in parallel may temporarily have an in-degree below k and out-degree l. This could result in the vertex being peeled more than once, and thus incorrect results. E.g. in Figure 3 v_1 and v_2 in parallel update u's in- and out-degrees ($k = 3$, $l = 2$).

The removal of edges (v_1, u) and (u, v_2) results in u having in-degree 2 (less than k) and out-degree 2 (equal to l). Thus, adding u to both frontiers $\mathcal{F}_0$ and $\mathcal{F}_1$ would lead to incorrect results. An atomic check on a vertex's peeling status before peeling it ensures that the vertex is added only to one frontier.

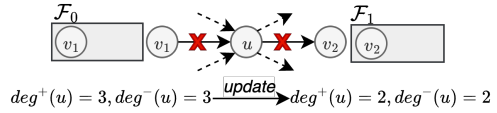


Fig. 3. Ensuring correct parallel peeling (race condition).

High atomic contention. Therefore, *the first limitation of peeling-based DCD on GPUs is the unavoidable contention caused by excessive atomic operations*, to remedy the issue of multiple threads trying to update the same vertex's degree simultaneously. This problem becomes particularly severe in real-world graphs where the presence of high-degree vertices with millions of neighbors leads to frequent atomic updates and, consequently, degrades performance.

Workload imbalance. When peeling a vertex, all its neighbors in at most (k,l)-core have to be inserted into the frontier. However, these frontier vertices can have very different degrees, from only a few neighbors to millions of neighbors. Thus, a single frontier contains vertices with vastly different workloads, making it difficult to design a workload balance strategy that benefits all vertices. For example, in `GPeel`, a block of threads shares the frontier, which allows updating a vertex's neighbors' degree at the thread, warp, or block level (Line 15, 18). However, *a key issue arises in determining the appropriate parallel granularity, since vertices in the frontier may exhibit highly diverse in- and out-degrees.* Assigning a thread to each peeled vertex is straightforward; however, due to the skewed degree distribution in real-world graphs, different threads may process neighbors of vastly different sizes, leading to significantly unbalanced workload among threads. This imbalance leads to divergent execution paths within a warp, as some threads will become idle waiting for the other threads in the warp to complete their operations. On the other hand, assigning an entire block of threads to a single vertex often results in thread under-utilization, since many vertices have only a small number of neighbors. A compromise is to assign a warp of threads (a warp = 32 threads) to each vertex in the frontier, allowing the neighbors of a vertex to be processed in parallel within the warp [1, 42]. Nevertheless, this strategy may still not be suitable for all types of real-world graphs. For example, a super node in a social network may have millions of neighbors, and a warp of threads may not provide enough parallelism to efficiently process all neighbors. In contrast, vertices with only a few neighbors may cause thread under-utilization, as most threads within the warp remain idle. Moreover, these degree updates generate the next batch of vertices to be peeled and insert them into the corresponding frontiers, whose degrees again exhibit large variation. This iterative dependency makes it difficult to control parallel granularity, resulting in persistent workload imbalance on GPUs.

To validate our argument, Figure 4 reports `GPeel`'s runtime under thread-, warp-, and block-level parallelism on six billion-scale real-world datasets (cf. Table 2). The results show that no single parallelization strategy consistently achieves the best performance due to varying degree distributions across graphs. Thread-level parallelism yields the lowest efficiency due to severe workload imbalance and warp divergence. Block-level parallelism performs best on H11 and H09, which have relatively high average degrees and moderate skew with balanced in/out-degree distributions. In contrast, on IT, IC, and EU, warp-level parallelism outperforms block-level assignment since these graphs have lower average degrees (e.g., IT: $d_{\text{avg}} = 27$) and highly skewed degree

distributions—conditions under which block-level mapping underutilizes threads, while warp-level mapping better matches the workload. These findings *motivate the design of a load-balanced algorithm that performs robustly across diverse graph structures.*

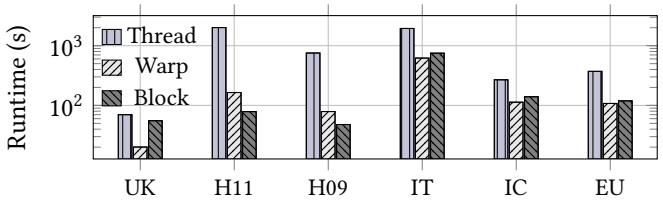


Fig. 4. GPeel thread-, warp-, block-level parallel runtime.

Redundant scans. Lastly, *peeling DCD suffers from excessive redundant scans of the entire vertex set*, an inefficiency that is specifically severe for dense graph structures. Consider the example in Figure 5, with in-degree constraint $k = 3$ and one frontier $\mathcal{F}$. GPeel scans the entire vertex set and peels the vertex with in-degree < 3 or out-degree $l = 0$, resulting in vertices a, b, c, f, i being peeled. Next, increase l to 1 and rescan the vertex set, but no vertex satisfies the condition of in-degree < 3 and out-degree $= 1$. The same holds for $l = 2$, where no vertex is peeled. In both cases, although yielding no peeled vertices, GPeel still needs to scan the whole vertex set (Algorithm 1, Line 5), which harms performance, particularly on large graphs. Finally, when $l = 3$, vertices d, e, g, h all have out-degree 3; they are peeled from the graph and added to the frontier.

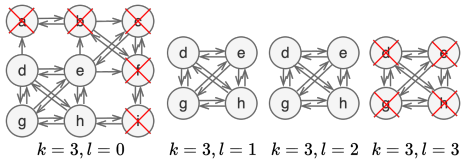


Fig. 5. Excessive redundant scan of the graph.

Discussion. GPeel incorporates key optimizations in state-of-the-art GPU k-core decomposition [1, 42]. First, it employs a block-level frontier strategy to distribute peeling tasks across thread blocks, enabling efficient utilization of GPU parallelism. Second, beyond warp-level neighbor updates, GPeel devises thread-level and block-level peeling strategies. Third, GPeel reduces the number of atomic operations by adopting optimized data structures. As such, GPeel is a competitive representative baseline. Despite GPU optimization, it still suffers from atomic contention due to concurrent updates, workload imbalance caused by skewed degree distributions in real-world graphs, and redundant scans during the peeling process.

6 Geld

We identified challenges in the GPU adaptation of vertex-dependent peeling for DCD. The other popular alternative for DCD, h-index-based approaches, maintains a per-vertex histogram whose size easily exceeds the memory capacity of consumer GPUs (typically, 12 - 24 GB). Moreover, updating these histograms requires scanning each vertex's neighbor set, which leads to workload imbalance.

We thus present a new algorithmic solution: Geld, GPU-accelerated load-balanced D-core decomposition for consumer GPUs. To address the workload imbalance issue across threads, we introduce a *degree-aware strategy* that adaptively assigns work unit sizes. For memory efficiency and reduced thread contention, we propose an *enumeration-based strategy* that eliminates auxiliary

data structures. Further, we design three pruning strategies, as well as a binary search warm-start strategy to reduce the number of iterations.

We recall the h-index [16] property that characterizes D-core.

Property 3 (h-index)**.** *If a vertex v belongs to the (k,l)-Core, i.e., $C_k[v] = l$, then v must have at least l out-neighbors and at least k in-neighbors whose out-core number is greater than or equal to l. We define $C_k[v]$ as the maximum integer l such that*

$$\begin{cases} |\{u \in N^-(v) \mid C_k[u] \geq l\}| \geq l, \\ |\{u \in N^+(v) \mid C_k[u] \geq l\}| \geq k. \end{cases} \tag{1}$$

Therefore, given in-degree constraint k and vertex v, we determine v's out-core number directly from its neighbors' out-core numbers using Eq. 1. `GeId` decouples the in- and out-neighbor constraints in Eq. 1 and obtains Eq. 2, where $\mathcal{H}_k^-[v]$ denotes the out-core number of v imposed by v's out-neighbors constraint, and $\mathcal{H}_k^+[v]$ refers to the out-core number of v imposed by v's in-neighbors constraint. To satisfy both in- and out-neighbor constraints, the out-core number of a vertex v is the minimum of $\mathcal{H}_k^-[v]$ and $\mathcal{H}_k^+[v]$. Thus, $C_k[v] = \min(\mathcal{H}_k^-[v], \mathcal{H}_k^+[v])$.

$$\begin{cases} \left|\{u \in N^-(v) \mid C_k[u] \geq \mathcal{H}_k^-[v]\}\right| \geq \mathcal{H}_k^-[v], \\ \left|\{u \in N^+(v) \mid C_k[u] \geq \mathcal{H}_k^+[v]\}\right| \geq k. \end{cases} \tag{2}$$

Table 1. Computing out-core number for $v \in V$ with in-degree $k = 0$ of Graph G in Figure 2.

V	a	b	c	d	e	f	g	h	i
$C_0[v] = \deg^-[v]$	1	3	2	4	6	3	3	4	2
$N^+(v) = \{u \in V \mid (u,v) \in E\}$	$\{b,d\}$	$\{a,e,f\}$	$\{b,e,f\}$	$\{e,g,h\}$	$\{d,g,h\}$	$\{b,c,e,i\}$	$\{d,e,h\}$	$\{d,e,g,i\}$	$\{f,h\}$
	$\{3,4\}$	$\{1,6,3\}$	$\{3,6,3\}$	$\{6,3,4\}$	$\{4,3,4\}$	$\{3,2,6,2\}$	$\{4,6,4\}$	$\{4,6,4,2\}$	$\{3,4\}$
$\mathcal{H}_0^+[v]$	$\min(1,4)$	$\min(3,6)$	$\min(2,6)$	$\min(4,6)$	$\min(6,4)$	$\min(3,6)$	$\min(3,6)$	$\min(4,6)$	$\min(2,4)$
$N^-(v) = \{u \in V \mid (v,u) \in E\}$	$\{b\}$	$\{a,c,f\}$	$\{e,f\}$	$\{a,e,g,h\}$	$\{b,c,d,f,g,h\}$	$\{b,c,i\}$	$\{d,e,h\}$	$\{d,e,g,i\}$	$\{f,h\}$
	$\{3\}$	$\{1,2,3\}$	$\{6,3\}$	$\{1,6,3,4\}$	$\{3,2,4,3,3,4\}$	$\{3,2,2\}$	$\{4,6,4\}$	$\{4,6,3,2\}$	$\{3,4\}$
$\mathcal{H}_0^-[v]$	$\min(1,1)$	$\min(3,2)$	$\min(2,2)$	$\min(4,3)$	$\min(6,3)$	$\min(3,2)$	$\min(3,3)$	$\min(4,3)$	$\min(2,2)$
$C_0[v] = \min(\mathcal{H}_0^+[v], \mathcal{H}_0^-[v])$	1	2	2	3	3	2	3	3	2

Example 4. *Table 1 calculates the out-core number with in-degree constraint $k = 0$ for all vertices (i.e., $C_0[\cdot]$) of Graph G in Fig. 2. We initialize the out-core number for each vertex v with its out-degree, as a vertex's out-core number cannot exceed its out-degree. Then, we calculate each vertex's largest $\mathcal{H}_0^+[\cdot]$ and $\mathcal{H}_0^-[\cdot]$ based on its in- and out-neighbors' out-core number $C_0[\cdot]$. Taking vertex b as an example, we compute its out-core number based on the in-degree constraint. For $k = 0$, vertex b must have at least zero in-neighbors whose out-core number is no smaller than $\mathcal{H}_0^+[b]$. In this case, $\mathcal{H}_0^+[b] = 6$ satisfies the condition, since b has an in-neighbor e with an out-core number of 6. At the same time, $\mathcal{H}_0^+[b]$ must be less than or equal to $C_0[b]$. Thus, $\mathcal{H}_0^+[b] = \min(3,6)$. Similarly, vertex b must have at least $\mathcal{H}_0^-[b]$ out-neighbors whose out-core numbers are no smaller than $\mathcal{H}_0^-[b]$. Given that the out-core numbers of b's out-neighbors are $\{1,2,3\}$, we obtain $\mathcal{H}_0^-(b) = 2$. Moreover, $\mathcal{H}_0^-[b]$ should not exceed $C_0[b]$. Therefore, the core number of b for $k = 0$ is $C_0[b] = \min(\mathcal{H}_0^+[b], \mathcal{H}_0^-[b]) = \min(3,2) = 2$.*

The example highlights a key challenge: how can we efficiently compute h-index values $\mathcal{H}_k^+[\cdot]$ and $\mathcal{H}_1^+[\cdot]$ for each vertex and for each in-degree k? We observe that it is not necessary to calculate the h-index values for all vertices, but only for those that are part of the (k,0)-core. The first pruning rule 1 harnesses exactly this hierarchical structure of the D-core.

Pruning Rule 1. *Given in-degree k, we can omit the h-index value calculation for these vertices that are not part of the (k, 0)-core.*

For a given k, computing $\mathcal{H}_k^+[v]$ and $\mathcal{H}_k^-[v]$ for vertices $v \in (k,0)$-core inevitably requires scanning both their in- and out-neighbors, since a vertex's out-core number depends on its neighbors' out-core numbers. However, this introduces the same thread workload imbalance seen in GPeel, due to variations in in- and out-degrees. To address this, Geld employs a *degree-aware strategy* that groups vertices with similar degrees into the same frontier. Each frontier follows a consistent computation pattern, ensuring more balanced workloads across threads and mitigating under- and over-utilization caused by skewed degree distributions. In contrast, peeling's workload is inherently harder to balance on GPUs: removing a vertex affects not only itself but also its neighbors, whose degrees can vary widely. Consequently, peeling frontiers contain vertices with highly heterogeneous degrees, making fine-grained load balancing infeasible.

Degree-aware strategy. Our degree-aware strategy adaptively assigns a different level of parallelism to each vertex according to its in- or out-degree and applies different strategies to compute $\mathcal{H}_k^+[\cdot]$ and $\mathcal{H}_k^-[\cdot]$. Algorithm 2 presents Geld. For each in-degree k, we place in parallel all vertices belonging to the $(k,0)$-core into the active set $\mathcal{A}$ and mark their convergence status as $\mathcal{M}[v]=1$, where 1 indicates that the vertex's out-core number is not yet finalized (Line 2). Then, we partition in parallel the unconverged vertices into multiple frontiers, grouping the ones with similar degree (Lines 5-6). This grouping aligns vertices within a frontier to follow similar computational patterns, leading to better workload balance across threads. Moreover, as the vertex partitioning by in-degree (Line 5) and by out-degree (Line 6) are independent tasks, in addition to paralleling into multiple frontiers, we also parallelize the execution of the in-degree and out-degree partitioning. Here, we present our partition design based on out-degree (for in-degree the computation is analogous). We propose three levels of parallelism aligned with the architectural features of GPUs. *Thread-level* frontiers $\{\mathcal{F}_{t,i}^-\}_1^p$: vertices with $\deg^-[v] \le \alpha$ are processed by a single thread. Since low-degree vertices dominate power-law graphs, we create multiple thread-level frontiers so that different thread blocks can process them concurrently and fully exploit GPU parallelism. *Warp-level* frontiers $\{\mathcal{F}_{w,i}^-\}_1^p$: vertices with $\alpha < \deg^-[v] \le \beta$ are assigned to a warp of 32 threads. Similar to the thread-level case, we use multiple warp-level frontiers to distribute the many medium-degree vertices across warps in different thread blocks. *Block-level* frontier $\mathcal{F}_b^-$: Vertices with $\deg^-[v] > \beta$ are rare but require substantial parallelism, so we group them into a single block-level frontier shared across all thread blocks. If $\mathcal{F}_b^-$ is further partitioned, the resulting frontiers may differ in size, causing substantial resource waste as a high-degree vertex is processed by an entire block of threads. The degree-aware strategy effectively achieves two goals: (i) ensuring sufficient parallelism for the massive number of low- and medium-degree vertices, and (ii) avoiding thread under-utilization when handling the few high-degree vertices while still providing adequate parallelism for the few high-degree vertices.

Architecture-driven parameter setting. While our degree-aware strategy balances the workload among threads, an important question is how to calibrate α and β. To this end, we introduce a data-agnostic method to select the best parameters based solely on the hardware architecture. Since vertices with degree $\le \alpha$ are processed by a single thread, α should not exceed warp size (32) to mitigate warp divergence; otherwise, a large α would cause threads within a warp to process vertices with very different degrees, leading to severe divergence and workload imbalance. An extreme case is $\alpha > \max(\deg_{\max}^+, \deg_{\max}^-)$ in which all vertices are processed at the thread level, and Geld degenerates to thread-level parallelism. Similarly, since a full thread block is used for vertices with degree larger than β, β should be at least the block size (up to 1024) to avoid thread underutilization. At the same time, β should not be too large, as a very large β value increases workload and induces imbalance among warp-level threads, degrading performance. In the extreme case where $\beta > \max(\deg_{\max}^+, \deg_{\max}^-)$ and α=0, all vertices are assigned to the warp-level thread,

and Geld degenerates to the warp-level parallelism. Moreover, when $\beta = 0$, all vertices are assigned to block-level processing, and Geld degenerates to block-level parallelism. Thus, as we also see in our experiments, α=32 and β =1024 yield good performance tradeoff.

Algorithm 2: Geld

Data: $G = (V, E)$
Result: C: The coreness for each vertex
1 **for** $k \leftarrow 0$ **to** $k_{\max}$ **do**
2 $\quad \mathcal{A} \leftarrow (k, 0)$-core, $\mathcal{M}[\cdot] \leftarrow \mathbb{1}_{(v \in \mathcal{A})}$ // Pruning Rule 1
3 $\quad C_k[\cdot] \leftarrow (k > 0)\ ?\ C_{k-1}[\cdot]\ :\ \deg^-[\cdot]$ // Pruning Rule 2
4 $\quad$ **while** $\exists \mathcal{M}[\cdot] = 1$ **do**
$\quad\quad$ // In parallel: Partition_In and Partition_Out
5 $\quad\quad \{\mathcal{F}^+_{t,i}\}^p_{i=0}, \{\mathcal{F}^+_{t,i}\}^p_{i=0}, \mathcal{F}^+_b \leftarrow$ Partition_In$(\mathcal{A}, \mathcal{M})$
6 $\quad\quad \{\mathcal{F}^-_{t,i}\}^p_{i=0}, \{\mathcal{F}^-_{t,i}\}^p_{i=0}, \mathcal{F}^-_b \leftarrow$ Partition_Out$(\mathcal{A}, \mathcal{M})$
$\quad\quad$ // In parallel: Computation $\mathcal{H}^+_k[\cdot]$ and $\mathcal{H}^-_k[\cdot]$
7 $\quad\quad \mathcal{H}^+_k[\cdot] \leftarrow$ hindex_In$(\{\mathcal{F}^+_{t,i}\}^p_{i=0}, \{\mathcal{F}^+_{t,i}\}^p_{i=0}, \mathcal{F}^+_b, C_k[\cdot])$
8 $\quad\quad \mathcal{H}^-_k[\cdot] \leftarrow$ hindex_Out$(\{\mathcal{F}^-_{t,i}\}^p_{i=0}, \{\mathcal{F}^-_{t,i}\}^p_{i=0}, \mathcal{F}^-_b, C_k[\cdot])$
9 $\quad\quad C_k[\cdot] \leftarrow \min(\mathcal{H}^+_k[\cdot], \mathcal{H}^-_k[\cdot], \mathcal{M})$
$\quad\quad$ // Degree-aware strategy update with Pruning Rule 3
10 $\quad\quad \mathcal{M} \leftarrow$ Update_In$(\{\mathcal{F}^+_{t,i}\}^p_{i=0}, \{\mathcal{F}^+_{t,i}\}^p_{i=0}, \mathcal{F}^+_b, \mathcal{H}^+_k[\cdot], \mathcal{H}^-_k[\cdot]) \cup$
$\quad\quad\quad$ Update_Out$(\{\mathcal{F}^-_{t,i}\}^p_{i=0}, \{\mathcal{F}^-_{t,i}\}^p_{i=0}, \mathcal{F}^-_b, \mathcal{H}^+_k[\cdot], \mathcal{H}^-_k[\cdot])$

Another challenge is efficient computation of the h-index for each vertex in every frontier without incurring additional memory overhead. Previous solutions use a histogram [22, 26, 42], where each vertex maintains an auxiliary array sized by its maximum out-core number, which can equal its degree in the worst case. For a given vertex v, the algorithm scans all its neighbors and increments the bucket entry corresponding to a neighbor's out-core number. Then, it scans the bucket in descending order to determine the largest possible h-index value that satisfies Eq. 2. However, this approach is ill-suited for GPUs: first, updating the bucket arrays in parallel causes severe thread contention, as multiple threads may attempt to update the same bucket entry simultaneously. Contention necessitates atomic operations, which degrade parallel efficiency. Second, maintaining per-vertex buckets on large graphs can easily exceed the global memory capacity of consumer GPUs. *On the other hand, modern GPUs, equipped with thousands of concurrent lightweight threads, are particularly apt to execute a large number of fine-grained and regular parallel tasks. [4, 19, 30, 32]* To align with the architectural characteristics of GPUs and reduce memory consumption overhead, we design an *enumeration-based strategy* to compute the h-index value for each vertex.

Enumeration-based strategy. As the in-degree constraint k increases, the out-core number of a vertex cannot increase by virtue of the D-core's hierarchical structure (Property 2). Thus, we can bound the out-core number of v with a value range as follows.

Pruning Rule 2. *For a vertex v in the $(k, 0)$-core, its out-core number lies in $[0, \deg^-(v)]$ if $k = 0$, and in $[0, C_{k-1}(v)]$ if $k \geq 1$.*

For each in-degree k, we initialize the core number of each vertex v with its upper bound $C_{k-1}[v]$ if $k > 0$, and its out-degree if $k = 0$ (Alg. 2, L. 3) and iteratively refine this bound until all vertices in $\mathcal{A}$ converge to their actual out-core number (Alg. 2, L. 4–10). So, instead of computing the h-index directly via a bucket structure, our *enumeration-based strategy* progressively tests all values from upper to lower bounds and returns the largest one satisfying Eq. 2, without maintaining any auxiliary, memory-intensive data structure.

Here, we present the calculation of $\mathcal{H}_k^-[\cdot]$ for each of the vertices in the block-level frontier $\mathcal{F}_b^-$ in Algorithm 3. The computation of $\mathcal{H}_k^+[\cdot]$ is analogous. In detail, we process all vertices in the block-level frontier $\mathcal{F}_b^-$ in parallel, with each vertex assigned to an entire block of threads (Algorithm 3, Line 1). Within a block, the threads concurrently scan the out neighbors of vertex v, where each thread is responsible for a disjoint subset of v's out-neighbors $N^-(v)$ and maintains a thread-local counter $\mathcal{L}_t$ (Algorithm 3, Lines 4-5). This counter maintains the number of neighbors u that belong to the $(k,0)$-core and satisfies the upper bound condition $C_k[u] \leq \mathcal{U}_v$. In the end, we aggregate the thread-local counters $\mathcal{L}_t$ and compare with the current candidate upper bound $\mathcal{U}_v$. If the value is less than that of the current upper bound, we decrement the enumerated candidate value $\mathcal{U}_v$ by 1 and repeat the process until it satisfies the constraint (Algorithm 3, Line 6-7). Moreover, the computation of each vertex's h-index in the in-direction ($\mathcal{H}_k^+[\cdot]$) and out-direction ($\mathcal{H}_k^-[\cdot]$) is also independent; thus, we compute both directions in parallel (Algorithm 2, Lines 7–8). Vertices in the thread-level frontiers $\{\mathcal{F}_{t,i}^+\}_{i=1}^p$, $\{\mathcal{F}_{t,i}^-\}_{i=1}^p$ and warp-level frontier $\{\mathcal{F}_{w,i}^+\}_{i=1}^p$, $\{\mathcal{F}_{w,i}^-\}_{i=1}^p$ adopt the same enumeration-based strategy, differing only in their parallel granularity—that is, each vertex in a thread- or warp-level frontier is assigned to a single thread or warp, respectively. All frontiers are processed concurrently, and within each frontier, vertices compute their h-index values in parallel.

Algorithm 3: Block-level $\mathcal{H}_k^-[\cdot]$ computation

```
   Data: F_b^-, C_k[·], k
   Result: H_k^-[·]
1  forall v ∈ F_b^- do                                              // In parallel
2      U_v ← C_k[v]                                                 // Pruning Rule 2
3      while U_v ≥ 0 do
           /* In parallel with a block of threads                                */
4          forall u ∈ N^-(v) do
5              L_t ← L_t + 𝟙_{u∈(k,0)-core ∧ C_k[u]≥U_v}
6      if BLOCK_REDUCE(L_t) ≥ U_v then H_k^-[v] ← U_v, break
7      else U_v ← U_v − 1
```

Example 5. *Our enumeration strategy, combined with pruning techniques 1, 2, reduces the number of iterations required for convergence. As shown in Fig. 2 with $k = 3$, we skip vertices a, b, c, f, i since they are not part of the $(3,0)$-core (pruning rule 1), and initialize the out-core numbers of d, e, g, and h with $C_2[\cdot] = 3$ (pruning rule 2). Starting from 3, our enumeration quickly satisfies the constraint for all vertices, allowing* `Geld` *to converge in* 1 *iteration, whereas* `GPeel` *requires* 4 *iterations, starting from out-degree constraint of 0, increasing to 3.*

Binary-search Warm Start. Our enumeration strategy begins with each vertex's upper bound and gradually decreases until it reaches its true h-index value. However, when $k = 0$, this progressive enumeration can be inefficient, since the upper bound is the vertex out-degree. Thus, it may iterate up to $\max_v(\deg^-[v] - C_0[v])$ iterations. To accelerate the initialization at $k = 0$, we replace the progressive approach with a binary search strategy to locate the largest value satisfying the h-index constraint faster.

Our strategy is GPU-friendly, as each thread performs *fine-grained* and *regular* tasks in parallel. Since each thread maintains its own local counter, our approach is completely *atomic-free* at the thread level. Moreover, each thread directly scans the out-core numbers of neighbors without the need to maintain an auxiliary bucket structure for each vertex. Finally, our strategy naturally

enforces uniform operations among all threads within the same warp, thereby avoiding thread divergence.

After computing $\mathcal{H}_k^+[v]$ and $\mathcal{H}_k^-[v]$ for each unconverged vertex ($v \in \mathcal{A}$, $\mathcal{M}[v] = 1$), we update the out-core number $C_k[v]$ in parallel as the minimum $\min(\mathcal{H}_k^+[v], \mathcal{H}_k^-[v])$ of the two values (Algorithm. 2, Line 9). However, a change in the out-core number of a vertex v may decrease the out-core numbers of its in- and out-neighbors. We observe that it is unnecessary to recompute the out-core numbers of all neighbors of v; only reconsider those potentially influenced vertices. This observation is incorporated in the following pruning rule.

Pruning Rule 3. *Let v be a vertex whose out-core number is updated to $C_k[v]$. Then, v can influence the out-core number of a neighbor u only if $C_k[v] < C_k[u]$.*

Following our degree-aware strategy, each vertex in the frontier marks its influenced neighbors $\mathcal{M}[u] = 1$ (i.e., those satisfying $C_k[u] > C_k[v]$) for the next round of computation. If there is no influenced vertex (i.e., $\forall \mathcal{M}[\cdot] = 0$), indicating all the vertices converge to their actual out-core number.

Example 6. *Consider the example in Figure 6. Before the out-core number update, vertex v contributes to both vertices u_1 and u_2 because $C_k[v] \geq C_k[u_1]$ and $C_k[v] \geq C_k[u_2]$. However, after the update, the out-core number of vertex v decreases to 3. Since the condition $C_k[v] \geq C_k[u_1]$ still holds, the reduction of v's out-core number has no effect on vertex u_1. In contrast, for vertex u_2, we now have $C_k[v] < C_k[u_2]$, which means u_2 may no longer have enough in-neighbors to satisfy the constraint in Equation 2. Thus, we mark u_2's convergence status to 1 and recompute its out-core number.*

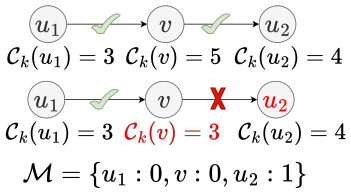


Fig. 6. Updating the influence vertex status $\mathcal{M}$.

Theorem 1. *For any in-degree k, Algorithm. 2 (*`Geld`*) converges in a finite number of iterations.*

Proof. Let $C_k[\cdot]_{(i)}$ denote the vector of out-core numbers at iteration i, and $\mathcal{H}_k^+[\cdot]_{(i)}$, $\mathcal{H}_k^-[\cdot]_{(i)}$ the h-index values imposed by the in- and out-neighbors, resp. By construction of Alg. 2, we have

$$\mathcal{H}_k^+[\cdot]_{(i)} \;\leq\; C_k[\cdot]_{(i-1)}, \quad \mathcal{H}_k^-[\cdot]_{(i)} \;\leq\; C_k[\cdot]_{(i-1)}.$$

Since $C_k[\cdot]_{(i)}$ is updated as the minimum of $\mathcal{H}_k^+[\cdot]_{(i)}$ and $\mathcal{H}_k^-[\cdot]_{(i)}$, it follows that

$$C_k[\cdot]_{(0)} \;\geq\; C_k[\cdot]_{(1)} \;\geq\; \cdots \;\geq\; C_k[\cdot]_{(t)}.$$

Thus, the sequence is monotonically non-increasing. Moreover, the out-core number of each vertex is a non-negative integer, so the sequence is bounded below by 0. Therefore, the algorithm must converge in finitely many iterations. □

Theorem 2. *For any in-degree k, Algorithm. 2 (*`Geld`*) converges to the correct out-core numbers of all vertices $v \in V$.*

Proof. For in-degree k, all vertices converge in iteration t with out-core number $C_k[\cdot]_{(t)}$. Assume there is a vertex $v \in V$ such that $C_k[v]_{(t)}$ is not the correct out-core number, i.e., Eq. 1 fails at v:

$$\left|\{u \in N^-(v) \mid C_k[u]_{(t)} \geq C_k[v]_{(t)}\}\right| < C_k[v]_{(t)} \quad \text{or}$$

$$\left|\{u \in N^+(v) \mid C_k[u]_{(t)} \geq C_k[v]_{(t)}\}\right| < k.$$

Thus, we have $\mathcal{H}_k^-[v]_{(t)}$ and $\mathcal{H}_k^+[v]_{(t)}$ value at iteration t,

$$\mathcal{H}_k^-[v]_{(t)} < C_k[v]_{(t)} \quad \text{or} \quad \mathcal{H}_k^+[v]_{(t)} < C_k[v]_{(t)}.$$

$$\text{Hence, } \min\{\mathcal{H}_k^-[v]_{(t)}, \mathcal{H}_k^+[v]_{(t)}\} \; < \; C_k[v]_{(t)},$$

which implies that another iteration would strictly decrease the out-core number value of vertex v. This contradicts termination at iteration t. Thus, Eq. 1 holds for all vertices in $C_k[\cdot]_{(t)}$, and the result is correct. □

Theorem 3. *Under the work-span model [17],* Geld *compute the D-core decomposition in* $|V| \cdot |E|$ work *and* $\sum_{k=1}^{k_{\max}} \Delta_k \cdot \log(|V|)$ *span.*

Proof. Each vertex v starts with $\deg^-(v)$ and eventually converges to $C_{k_{\max}}[v]$, with each h-index calculation requiring scanning both its in- and out-neighbors. Let $\deg[v] = (\deg^+[v] + \deg^-[v])$ Thus, the total work is

$$\sum_v (\deg^-[v] - C_{k_{\max}}[v]) * (\deg[v]) \leq \sum_v (\deg^-[v]) * (\deg[v])$$

Let $\deg_{\max} = \max_v(\deg[v])$ be the maximum total degree over all vertices. For any vertex v, we have $(\deg^+[v] + \deg^-[v]) \leq \deg_{\max}[v] \leq 2 \cdot |V|$. Therefore,

$$\sum_v (\deg^-[v]) * (\deg[v]) \leq \deg_{\max} \cdot \sum_v (\deg^-[v]) \leq |V| \cdot |E|$$

Let $\Delta_k = \max_v(C_{k-1}[v] - C_k[v])$ be the maximum decrease in the out-number of any vertex from in-degree $k-1$ to k. Since Geld perform $k_{\max}$ iterations, the total number of iterations is $\sum_{k=1}^{k_{\max}} \Delta_k$. In each iteration, every vertex v performs a reduction over its in- and out-neighbors to count how many neighbors will satisfy the threshold condition. This reduction is bound by $\log(|V|)$. Therefore, the overall span is $\sum_{k=1}^{k_{\max}} \Delta_k \cdot \log(|V|)$. □

Discussion. Our Geld effectively addresses the challenges of scalable graph algorithms on GPUs (Section 4). First, the degree-aware strategy groups vertices with similar degrees to follow the same operations, leveraging thread-, warp-, and block-level parallelism to mitigate workload imbalance (**C3**). Second, the enumeration-based strategy replaces the computation of $\mathcal{H}^+[\cdot]$ and $\mathcal{H}^-[\cdot]$ via per-vertex histograms with a lightweight enumeration scheme, thereby avoiding histogram maintenance and alleviating the limited thread- and shared-memory bottleneck (**C2**). Finally, by combining these two strategies, Geld exploits the fast thread-level memory and ensures that all threads within the same thread-, warp-, or block-frontier execute uniform local counting operations, effectively eliminating thread divergence (**C1**).

D-core maintenance. Real-world graphs are often dynamic with edge insertions and deletions. In such cases, we can easily extend Geld to support dynamic graphs by incorporating existing maintenance strategies [13]. Specifically, given an edge insertion or deletion (u, v), Geld first computes the $k_{\max}$ values of vertices u and v, i.e., the largest k such that they belong to the $(k, 0)$-core. Then, for each $k \in \{0, \min(k_{\max}(u), k_{\max}(v))\}$, we run Geld to recompute out-core numbers for vertices within the corresponding $(k, 0)$-core. As dynamic updates only affect a small range of in-degree constraints, maintenance can be performed by restricting computation to only a few k values rather than full recomputation.

7 Experimental Evaluation

We implement our GPU algorithms in CUDA and run the experiments on an NVIDIA GeForce RTX 4090 with 24GB of memory and compile with nvcc version 12.2.14. We configure our GPU algorithms kernel with $p = 256$ blocks and $\mathcal{T} = 1024$ threads per block. Our implementation is publicly available [1]. For the state-of-the-art, we obtain each algorithm's C++ implementation and perform experiments on an Intel® Xeon® Silver 4410T processor with 40 threads and 252GB of RAM, using OpenMP and the GCC 11.4.0 compiler with the -O3 optimization.

Data. We perform experiments on **12** real-world datasets [5, 6, 21] [2] from various domains. Table 2 presents their characteristics.

Table 2. Graph properties: number of nodes |V|, number of edges |E|, maximum in-degree $\deg^+_{\max}$, maximum out-degree $\deg^-_{\max}$, maximum in-core number $k_{\max}$, and domain.

Graph	|V|	|E|	$\deg^+_{\max}$	$\deg^-_{\max}$	$k_{\max}$	Domain
(AM)Amazon	0.4M	3.20M	2.7K	10	10	Product
(EM)Email-EuAll	0.27M	0.42M	7.6K	929	27	Communication
(PO)Pokec	1.63M	30.62M	13.7K	8.8K	32	Social
(SD)Slashdot	82.17K	0.87M	2.5K	2.5K	53	Social
(EN)Enwiki-2024	6.8M	0.17B	0.24M	13.1K	104	Social
(LJ)Live Journal	4.85M	68.48M	13.9K	20.3K	252	Social
(UK)UK-2002	18.5M	0.29B	0.19M	2.5K	942	Web
(H11)Hollywood-2011	2.18M	0.23B	13.1K	13.1K	1297	Actor
(H09)Hollywood-2009	1.1M	0.11B	11.5K	11.5K	2208	Actor
(IT)IT-2004	41.29M	1.14B	1.32M	9.9K	3198	Web
(IC)IndoChina-2004	7.4M	0.2B	0.26M	6.9K	6821	Web
(EU)EU-2015-TPD	6.7M	0.17B	74.1K	0.4M	9568	Web

`GHindex`. We propose `GHindex`, a competitive baseline that incorporates the pruning techniques of `Geld` to reduce computational overhead. As in prior work, `GHindex` adopts a block-level frontier strategy to expose GPU parallelism [1, 42] and computes the h-index for each vertex using a per-vertex histogram and suffix scan [42]. For in-depth evaluation, we implement *four* `GHindex` variants with increasing parallelism. The thread-level variant assigns one thread per vertex and stores histograms in global memory due to limited thread-local storage, resulting in a memory footprint proportional to $|V| + |E|$ [42]. As each histogram is processed by a single thread, no contention occurs, enabling a compact `uint16_t` representation. In contrast, the warp- and block-level variants compute h-indices cooperatively, incurring thread contention, and therefore use `int`-based histograms in global memory. The shared-memory block-level variant further reduces memory access latency by maintaining per-block histograms in shared memory.

Algorithms. We compare `Geld`, with four state-of-the-art algorithms and the GPU peeling and H-Index algorithm. We use the public implementation [3] of `Peeling`, `SC`, `ParPeel`, and `Shell` from [26].

(1) `Peeling` [13]: State-of-the-art sequential DCD algorithm.
(2) `SC` [22]: State-of-the-art distributed DCD algorithm.
(3) `ParPeel` [26]: State-of-the-art parallel peeling DCD algorithm on multi-core CPU.
(4) `Shell` [26]: State-of-the-art parallel DCD algorithm on multi-core CPU.

[1] https://github.com/kouteisang/DCoreGPU
[2] https://snap.stanford.edu/data/ https://law.di.unimi.it/index.php
[3] https://github.com/GearlessL/PDC

(5) GPeel: Peeling-based DCD algorithm on GPU (Section 5).
(6) GHindex: H-Index-based DCD algorithm on GPU (Section 6).
(7) Geld: Our GPU-accelerated enumeration-based load-balanced DCD algorithm. We set $\alpha = 32$ and $\beta = 1024$ (Section 6).

To ensure a fair comparison with the state-of-the-art Shell [26], our GPU algorithms adopt its pruning strategy [26], which removes redundant computations when vertices have identical out-core numbers under different in-degree constraints k.

7.1 Runtime

We study the runtime of all algorithms for DCD, as an average over 5 runs. Please note the y-axis in a log scale for the following figures.

Peeling algorithms. We initially compare the runtime of our Geld with the peeling algorithms in Figure 7. We first compare Geld with GPeel. We report the best of breed GPeel among thread-, warp- and block-level parallelism. Under the same GPU setting, Geld achieves up to 81.3× speedup over GPeel, with 31× speedup on average, particularly on large graphs, even though GPeel is the top performer among peeling algorithms. Unlike GPeel, which increases the out-core number starting from 0 and incrementally for each in-degree k, Geld utilizes neighbor out-core numbers for faster convergence in much fewer iterations on most of the datasets in Table 3. However, when both $k_{\max}$ and $l_{\max}$ are small, GPeel finishes in fewer iterations. Moreover, our binary search warm start strategy is effective on the EU dataset, with ~ 28.8× speedup over the top-bottom approach.

We then compare Geld with the state-of-the-art serial Peeling and parallel ParPeel peeling algorithms; Geld achieves up to 5340× speedup over Peeling, with an average speedup of 1440×, and more than 2238× speedup over ParPeel. Even though ParPeel improves over serial Peeling on small datasets, it fails on billion-scale large datasets, and cannot complete in 5 hours on IT, IC, and EU, due to increasing thread contention and imbalanced workload between threads, while Geld effectively addresses these issues.

Table 3. Total # iterations for Geld and GPeel to compute DCD.

	AM	EM	PO	SD	EN-24	LJ
Geld	366	**220**	2824	**568**	**5159**	**5366**
GPeel	**121**	780	**1036**	2862	11294	45202
	UK-02	HW-11	HW-09	IT-04	IC	EU
Geld	**2066**	**943**	**725**	**8558**	**3540**	**3602**
GPeel	269698	1098108	1473403	2373658	3417812	3416349

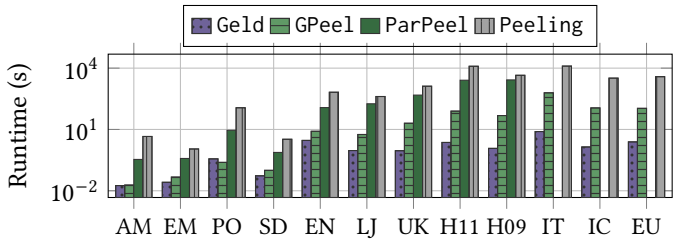


Fig. 7. Runtime of Geld, GPeel, ParPeel, and Peeling.

H-Index algorithms. We first compare Geld with GHindex under different levels of parallelism in Figure 8. The thread-level variant shows the lowest efficiency across datasets. Although it

avoids thread contention, workload imbalance across threads hampers its performance. The shared-memory block-level variant modestly improves upon the block-level design and cannot be applied to the EN, LJ, H11, and EU datasets due to their out-degrees exceeding the available shared memory. Thus, the global memory histogram design is unavoidable. Overall, no single parallelism strategy consistently outperforms the others, due to the diverse degree distributions of real-world graphs. In comparison, Geld achieves consistent average speedups of 24.9×, 6.33×, and 6.02× over the thread-level, warp-level, and block-level variants of GHindex.

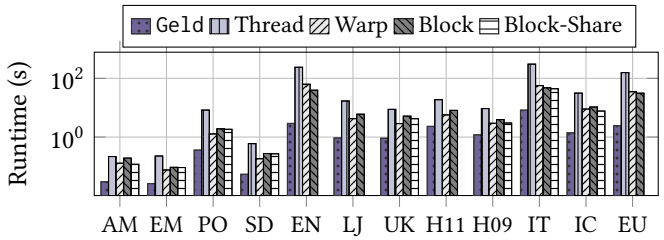


Fig. 8. Runtime of Geld and four variants of GHindex.

We then juxtapose with other H-Index algorithms with Geld in Figure 9. Geld is up to 47.4× faster than the state-of-the-art parallel algorithm Shell with an average of 22.1× speedup, and at least three orders of magnitude faster compared with the state-of-the-art distributed algorithm SC, especially on large datasets where SC cannot finish in 5 hours. Although all of Geld, Shell, and SC use the h-index property of D-core, Geld consistently outperforms the others on all datasets. This reflects the smart design of our algorithm that comprises the upper bound of each vertex's out-core number to thread-level, warp-level, and block-level search, the workload balancing, and a sapient use of thread-local memory to avoid thread contention. We also test the state-of-the-art Shell on a dual-socket Intel® Xeon® Silver 4316 server with 80 threads; however, the results show that increasing the computational resources actually leads to 1.3× to 43× slower performance, as the design of Shell does not account for the non-uniform memory access (NUMA) architectures. These results indicate that Shell exhibits limited scalability on multi-thread systems, where increasing the number of CPU threads does not yield proportional speedups and may even cause performance degradation. This observation highlights that merely increasing the thread counts does not necessarily improve performance. The performance gain of Geld on GPU stems from its GPU-friendly algorithmic design, rather than simply from additional computational threads.

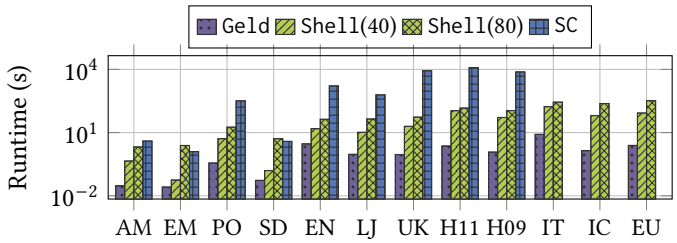


Fig. 9. Runtime of Geld, Shell and SC.

Degree-aware strategy. We demonstrate the effectiveness of our degree-aware strategy in balancing the workload across threads and improving overall efficiency by comparing Geld with three variants Geld-Thread ($\alpha, \beta \geq \max(\deg^-_{\max}, \deg^+_{\max})$), Geld-Warp ($\alpha$=0 and $\beta \geq \max(\deg^-_{\max}, \deg^+_{\max})$), and Geld-Block ($\alpha, \beta$=0). All four algorithms adopt the same enumeration-based strategy and the same number of GPU threads. The difference lies in their parallelization granularity: Geld-Thread assigns one thread to compute the h-index of each vertex, Geld-Warp assigns one warp of threads,

and Geld-Block assigns one block of threads. In contrast, our Geld employs a degree-aware strategy that adaptively selects the appropriate granularity based on each vertex's in- and out-degree. As shown in Figure 10 (Top), Geld achieves up to 38.8×, 6.1×, and 4.3× speedup over Geld-Thread, Geld-Warp, and Geld-Block, respectively, with average speedups of 11.3×, 2.4×, and 3.3×. This speedup is noticeable particularly on graphs with highly skewed degree distributions, such as EN, IT, IC, and EU.

In detail, Figure 10 (Bottom) reports the runtime of computing $\mathcal{H}^+[\cdot]$ on the EN and IT datasets, both of which exhibit highly skewed in-degree distributions. With our degree-aware strategy, Geld completes in only 1.5 seconds on EN and 1.9 seconds on IT, yielding 54.13× and 42.73× speedups over Geld-Thread, 7.5× and 3.7× over Geld-Warp, and 5× and 4.4× over Geld-Block, respectively. These results show that the thread-level parallelism suffers from a severe workload imbalance issue, where some threads are assigned to high-degree vertices with millions of degrees, while others process vertices with only a few, leading to the worst overall performance. In contrast, block-level parallelism assigns an entire block of threads to each vertex, increasing the parallelism for high-degree vertices but wasting substantial computational resources on low-degree vertices. Our degree-aware strategy alleviates both issues by adaptively matching the parallelism to the vertex degree, thereby improving workload balance, avoiding wasting computational resources, and further improving the computational efficiency.

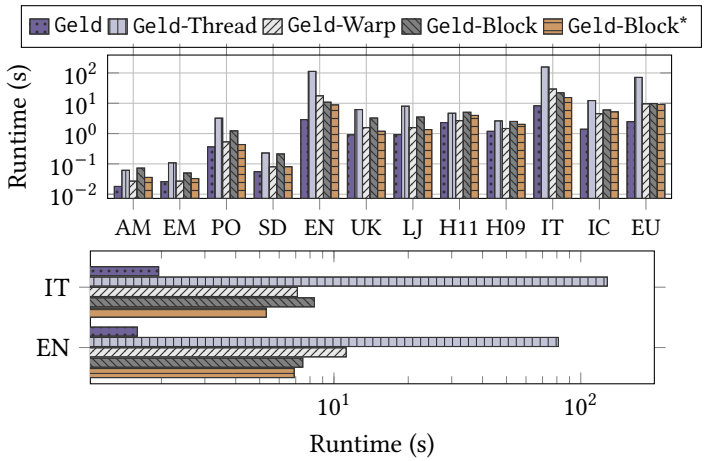


Fig. 10. Runtime of Geld variants (top), of $\mathcal{H}^+[\cdot]$ (bottom).

Additionally, in Geld, instead of dividing the block-level frontier into multiple parts, all thread blocks share a single frontier (Algorithm 2, Line 5-6). Splitting it into multiple parts often leads to imbalance, as each part may contain a different number of vertices, causing uneven workloads across threads. As shown in Figure 10, this shared-frontier strategy in Geld achieves more than a 3.8× speedup over the multi-block frontier partitioning baseline Geld-Block*. In particular, it achieves 2.73× and 4.32× speedup for computing $\mathcal{H}^+[\cdot]$ for IT and EN datasets, separately.

Parameter Sensitivity. Figure 11 examines the impact of parameters α and β on the performance of Geld. We fix α=32 and vary $\beta \in \{1024, 32768, 983040\}$ (top), and fix β=4096 while varying $\alpha \in \{32, 512, 2048\}$ (bottom). The parameter β affects highly skewed datasets with large $\deg^+_{\max}$ or $\deg^-_{\max}$ (i.e., EN, IT, IC, EU), where larger values significantly degrade performance. For instance, on EN, Geld with β=1024 is about 4.7× faster than with β=983040. Increasing β shifts more high-degree vertices to warp-level processing, exacerbating load imbalance: execution becomes dominated by warp-level threads, while block-level threads are underutilized. In contrast, β has little impact on

datasets with small degrees. Increasing α degrades performance. On average, Geld with α=32 is about 2× faster than with α=2048. Larger α values assign more work to thread-level processing, increasing per-thread workload and reducing warp-level utilization, which again leads to load imbalance.

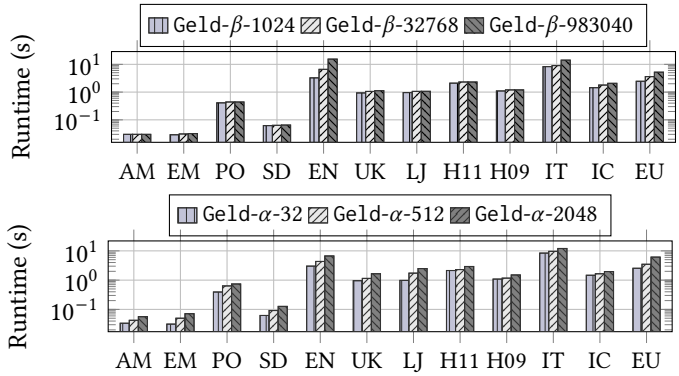


Fig. 11. Geld runtime varying β (top); varying α (bottom).

Pruning Rules. Figure 12 reports the runtime of Geld with three ablation baselines that disable the corresponding pruning rules. **Prn-1:** For each in-degree k, instead of restricting the computation space to (k,0)-core, Obs-1 computes the h-index value for all vertices, resulting in wasting computational resources, with Geld achieving up to 12.1× speedup, and 4.4× speedup on average over Prn-1. **Prn-2:** The second pruning rule reduces the enumeration space by computing the h-index value from its previous value as an upper bound. In contrast, Prn-2 always calculates the h-index value from its out-degree instead of relying on the tighter upper bound. Consequently, Geld achieves an average 261.8× speedup. By contrast, under Prn-2, the EU dataset fails to finish within 5 hours. **Prn-3:** Our last pruning rule restricts updates to vertices whose out-core number may change. Prn-3 recalculates the out-core number for all neighbors instead of the affected ones. Thus, Geld achieves up to 4.4× speedup with an average of 1.53×. Overall, these ablation studies highlight the effectiveness of our pruning rules in reducing the computation space and improving efficiency.

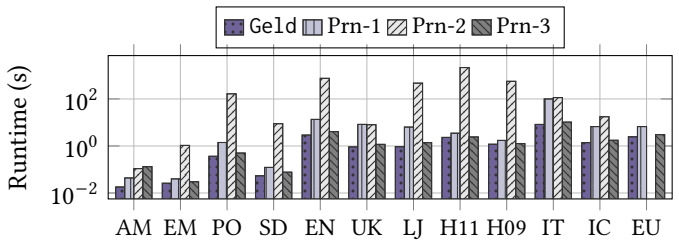


Fig. 12. Runtime of Geld with three pruning techniques.

Scalability with edge density. To study the scalability of Geld with edge density, we randomly sample 20%, 40%, 60%, 80%, and 90% of the edges from each dataset, inducing five sparser subgraphs of these respective edge percentages. Fig. 13 presents the runtime across these subgraphs. On small graphs such as AM, EM, PO, and SD, GPeel achieves performance comparable to or even slightly better than Geld, since these graphs have very small $l_{\max}$ and therefore require only a few iterations. Meanwhile, Geld incurs additional overhead from maintaining multiple frontiers. However, as the graph size and density increase, a larger $l_{\max}$ requires GPeel to run substantially more iterations, while the higher density results in increasing thread contention, and an imbalanced workload

among threads. As a result, the performance gap between GPeel and Geld widens. Similarly, Geld consistently outperforms GHindex as the graph density increases. In contrast, Geld consistently achieves the highest efficiency, superior to all other methods, particularly on large-scale graphs, achieving an average 26.76× speedup on IC-2004 and 27.87× speedup on EU-2015-TPD, over state-of-the-art Shell.

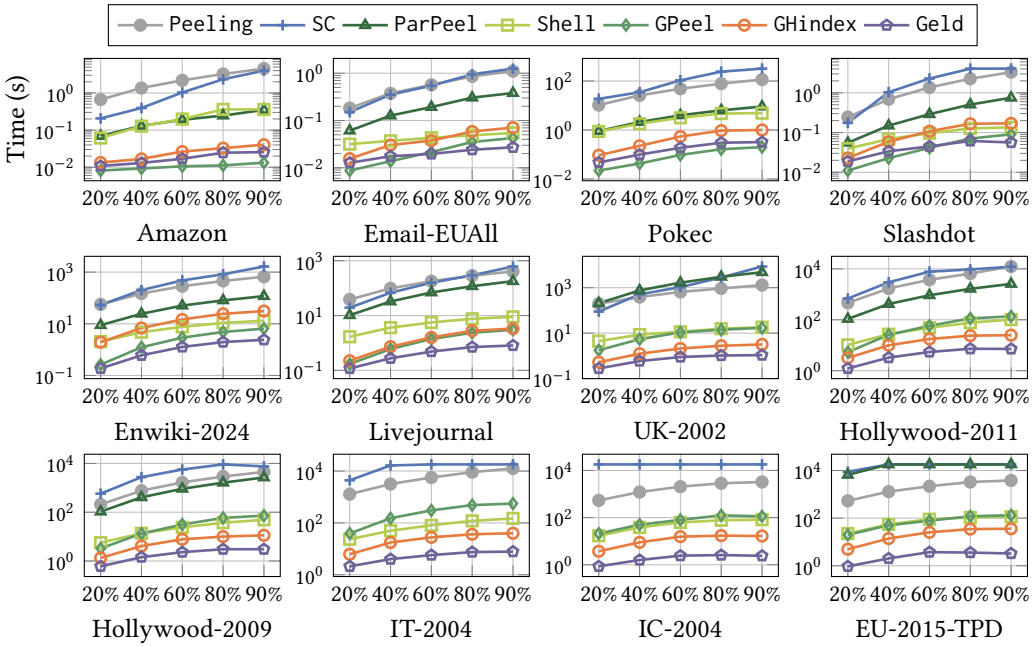
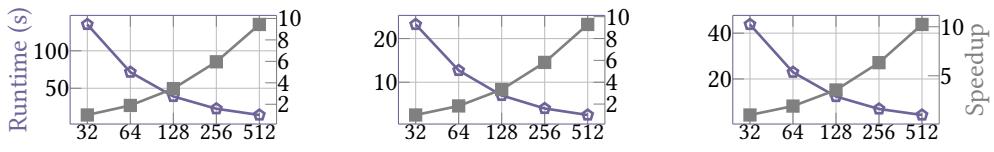


Fig. 13. Runtime of the D-Core decomposition algorithm with varying edge percentages. The y-axis is in log scale.

Scalability in threads. We study the scalability of our Geld with respect to the number of threads per block. We vary the number of threads per block among {32, 64, 128, 256, 512}. Figure 14 shows the scalability of our algorithms across various thread configurations, along with the corresponding speedup *relative to using 32 threads* per block. Due to space limits, we only report for the large-scale graphs, IT, IC, and EU, but observe a similar trend on other datasets. As the number of threads per block increases, Geld achieves over 10× speedup compared to 32-threads-per-block.



(a) IT-2004 (b) IC-2004 (c) EU-2015-TPD

Fig. 14. Runtime and speedup with the number of threads per block. Speedup relative to 32 threads per block.

The scalability stems from the degree-aware design of Geld. Geld partitions thread-level and warp-level frontier into p parts (Algorithm 2, Line 5-6). Assume allocates $\mathcal{T}$ threads to each part of the frontier; therefore, $\mathcal{T} \cdot p$ vertices work in parallel across thread-level frontiers, $\frac{\mathcal{T}}{32} \cdot p$ vertices work

in parallel across warp-level frontiers, and p vertices work in parallel across the block-level frontier. Therefore, with more threads per block, Geld can process more vertices in parallel within each batch. Moreover, Geld leverages a full block of threads for high-degree vertices, which also benefits from increased thread count per block. Additionally, in Geld, our enumeration strategy allows each thread to maintain its own register-level local count, and each warp/block performs a warp-/block-shuffle reduction to aggregate these local counts, effectively avoiding thread contention. In contrast, ParPeel suffer from increased thread contention as the number of threads grows, which leads to degraded performance.

D-core maintenance. We study DCD in dynamic graphs by generating five consecutive snapshots, each obtained by progressively inserting a random edge. Figure 15 reports the average runtime of full DCD recomputation using Geld (Geld-Full) and the maintenance variant Geld-Dyn (end of Section 6). Geld-Dyn achieves a 3.52× speedup over full recomputation, demonstrating that Geld efficiently supports DCD on dynamic graphs. Similar results for deletion omitted here due to space limitations.

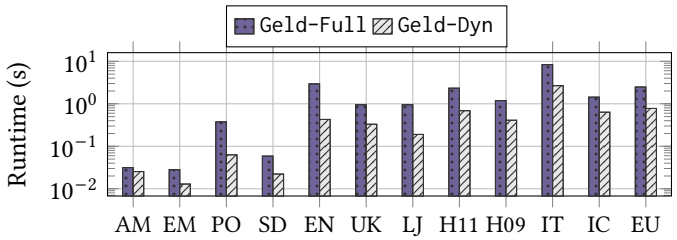


Fig. 15. Runtime Geld vs. dynamic Geld maintenance.

GPU Occupancy. GPU occupancy is a key metric for evaluating how effectively a GPU kernel utilizes hardware resources. Achieved occupancy is the average ratio of active warps, while the theoretical occupancy is the maximum ratio determined by kernel resource usage and hardware limits. Figure 16 shows that Geld consistently achieves high occupancy across datasets, with an average achieved occupancy of 62.5%. Moreover, the achieved-to-theoretical occupancy ratio exceeds 93.9% on average, which indicates our Geld effectively utilizes the available GPU resources.

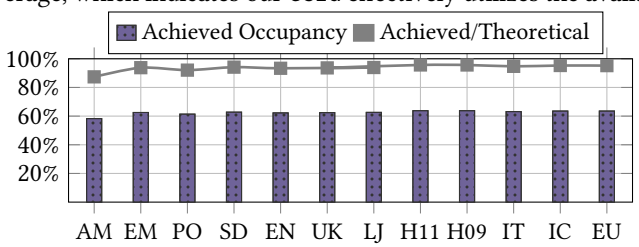


Fig. 16. Achieved and Theoretical GPU Occupancy of Geld.

7.2 Memory Consumption

Figure 17 compares the peak GPU memory consumption of Geld, GPeel, and GHindex, with breakdown in Table 4, including the CSR representation of the directed graph G, the frontier structures $\mathcal{F}_{i=0}^{p}$ used by GPeel and GHindex, and the histogram (Hist) maintained by GHindex. To reduce memory usage, GHindex reuses a single histogram for both in- and out-directions, halving the histogram overhead. The thread-level variant further employs a compact uint16_t histogram, whereas the warp- and block-level variants require int histograms due to atomic-update contention, which is unsupported for uint16_t in CUDA. Consequently, these variants incur twice the histogram memory of the thread-level variant.

In both Geld and GPeel, we allocate $\frac{|V|}{p}$ entries per part of the frontier on each dataset. GPeel maintains a single-level frontier, while Geld maintains three-level frontiers to balance the workload among threads. Thus, Geld reserves a total frontier space around $3 \cdot |V|$. With this minimal memory overhead, Geld delivers substantial performance gains with over 81.3× speedup over GPeel.

Compared with thread-level GHindex, Geld reduces memory consumption by 11% on average, and by 33% compared with warp-level and block-level variants. Although the thread-level variant of GHindex uses the least memory, it suffers from poor performance; on average 3.75× and 3.14× slower than warp-level and block-level variants, respectively. On large-scale graphs, the memory overhead of GHindex is particularly pronounced. For example, on IT, GHindex incurs an additional 2 246 MB to 4 492 MB of memory due to its per-vertex histogram of size $|V| + |E|$, resulting in a total GPU memory footprint of 13.1 GB to 15.14 GB, which exceeds the limits of affordable consumer GPUs such as RTX 3080 Ti (12 GB). In contrast, Geld leverages its enumeration-based strategy to avoid auxiliary data structures, thereby remaining well within the memory limits of consumer GPUs while delivering significantly higher performance.

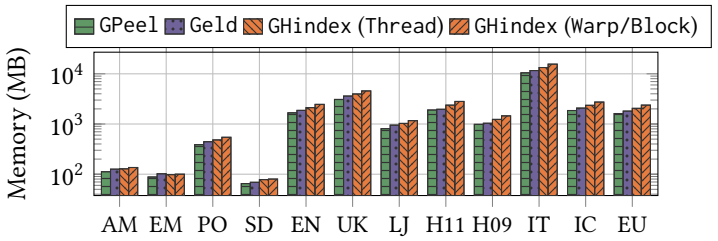


Fig. 17. Peak memory consumption of GPU algorithms.

Table 4. Breakdown of memory consumption (MB).

	AM	EM	PO	SD	EN	LJ	UK	HW11	HW09	IT	IC	EU
G	36	12	268	10	1424	604	2520	1780	888	9300	1584	1372
$\{\mathcal{F}_i\}_{i=0}^{p}$	1.6	1.1	6.5	0.3	27.2	19.4	74	8.7	4.4	165.2	29.6	25.6
Hist(uint_16t)	8	2	62	2	344	140	594	442	218	2246	380	330
Hist(int)	16	4	124	4	688	280	1188	884	436	4492	760	660

7.3 Application Studies

We here analyze algorithm design and real-world applications.

k-core decomposition. K-core decomposition finds the largest subgraph where every vertex has degree at least k. This type of cohesive subgraph mining is a well-studied technique in undirected graphs. Surprisingly, even this simpler, extensively analyzed algorithm benefits from our degree-aware and enumeration-based strategies. Across six real-world datasets (Table 5 [5, 6, 21]), Geld outperforms the state-of-the-art h-index-based GPU algorithm HistoCore [42], achieving 6.1% to 47.9% improvement over its bucket-based approach (Figure 18). This case study demonstrates that our strategies yield clear, measurable gains beyond DCD, though broader evaluation lies outside the scope of this work.

Social Network Analysis. Prior research suggests that influential spreaders are often located in the *innermost core* of a network [20], defined as the D-core with the largest in-degree k and the corresponding largest out-degree l. Identifying such core structures enables effective strategies for influence maximization and rumor containment in social networks. As networks evolve, the individuals in the innermost core change, offering valuable insights into shifting influence. As an

Graph	|V|	|E|
Livejournal	4.85M	42.85 M
Hollywood2009	1.2M	56.38M
Orkut	3.07M	117.18M
Indochina2004	7.4M	150.98M
UK2005	39.46M	798.05M
IT2004	41.29M	1.03B

Table 5. Data Properties

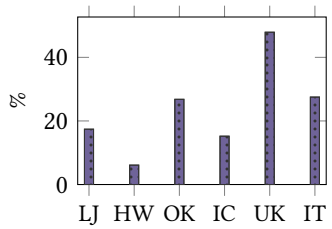


Fig. 18. Performance Gain.

illustrative study, we analyze CollegeMsg, a temporal network of private messages at UC Irvine from April to October 2004 [21]. Each record is a triplet (u, v, t), where user u sends a message to user v at time t, forming a directed edge. To study influence dynamics, we apply `Geld` to two periods: before August 15, 2004, and before October 26, 2004. This allows us to examine how the innermost core user evolves between August and October, specifically, which users become more active and join the innermost core, and which users become less active and drop out.

The innermost core has $k_{\max} = 14$ for both time frames, while the maximum l increases from 8 to 11, indicating more communication between users from August to October. Figure 19 visualizes the evolution of the innermost core over time. The purple cluster in the center is the users who remain in the innermost core throughout both periods. The 63 blue nodes on the left represent users who belonged to the $(14, 8)$-core but disappeared in the later $(14, 11)$-core, implying a decline in their activity during the latter months. The 6 green nodes on the right highlight users who were absent in $(14, 8)$-core but emerge in $(14, 11)$-core, indicating their increased activity between August and October. This case study illustrates how D-core decomposition can help detect interaction behavior in a social network and identify the innermost core. As most of the real-world social networks are very large, our efficient D-core decomposition algorithm can rapidly detect such changes.

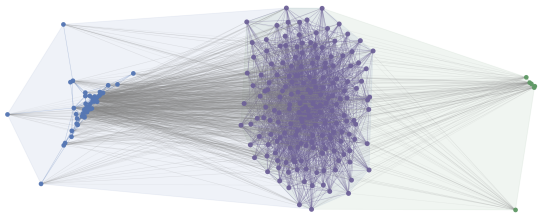


Fig. 19. Innermost core analysis over time (left to right).

8 Conclusion

This paper presents an efficient, scalable solution for D-core decomposition on GPUs. We address thread divergence, limited memory, and load balancing issues on the GPU for fast D-core computation as a fundamental approach for analyzing directed graphs. Our investigation of GPU peeling algorithms finds three major issues: high atomic contention, workload imbalance, and redundant scans. To address these, our `Geld` offers a degree-aware strategy to achieve thread workload balance, and an enumeration-based strategy to achieve memory efficiency. Extensive experimental evaluation on 12 real-world datasets shows substantial performance gains, with `Geld` achieving an average speedup of 31× over GPU peeling DCD, 6× over GPU H-Index DCD, 22× over state-of-the-art parallel D-core decomposition, as well as 2× memory efficiency, and 3 orders of magnitude speed-up over the best serial algorithm.

Acknowledgments

The research is funded by Aarhus University Research Foundation (grant 2022-0142).

References

[1] Akhlaque Ahmad, Lyuheng Yuan, Da Yan, Guimu Guo, Jieyang Chen, and Chengcui Zhang. 2023. Accelerating k-Core Decomposition by a GPU. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1818–1831.

[2] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. k-core decomposition: A tool for the visualization of large scale networks. *arXiv preprint cs/0504107* (2005).

[3] Vladimir Batagelj and Matjaz Zaversnik. 2003. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[4] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW 2011*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). 587–596.

[6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW 2004*. 595–601.

[7] Lijun Chang and Lu Qin. 2019. Cohesive subgraph computation over large sparse graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2068–2071.

[8] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*. 3544–3550.

[9] Yiran Cheng, Xibo Sun, and Qiong Luo. 2024. Rapidgkc: Gpu-accelerated k-mer counting. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3810–3822.

[10] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. Park: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE international conference on big data (Big Data)*. IEEE, 9–16.

[11] Mario Eboli. 2010. An algorithm of propagation in weighted directed graphs with applications to economics and finance. *International Journal of Intelligent Systems* 25, 3 (2010), 237–252.

[12] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2022. *Cohesive subgraph search over large heterogeneous information networks*. Springer.

[13] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 11 (2018), 2093–2107.

[14] David Garcia, Pavlin Mavrodiev, Daniele Casati, and Frank Schweitzer. 2017. Understanding popularity, reputation, and social influence in the twitter society. *Policy & Internet* 9, 3 (2017), 343–364.

[15] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems* 35, 2 (2013), 311–343.

[16] J. E. Hirsch. 2005. *H-index*. Accessed: 2025-10-17.

[17] Joseph JáJá. 1992. *Parallel algorithms*.

[18] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1482–1491.

[19] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE micro* 31, 5 (2011), 7–17.

[20] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888–893.

[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[22] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core decomposition over large directed graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1546–1558.

[23] Hwijoon Lim, Juncheol Ye, Sangeetha Abdu Jyothi, and Dongsu Han. 2024. Accelerating model training in multi-cluster environments with consumer-grade gpus. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM)*. 707–720.

[24] Dandan Liu, Run-An Wang, Zhaonian Zou, and Xin Huang. 2024. Fast multilayer core decomposition and indexing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2695–2708.

[25] Youzhe Liu, Xiaojun Dong, Yan Gu, and Yihan Sun. 2025. Parallel k-core decomposition: Theory and practice. *Proceedings of the ACM on Management of Data (SIGMOD)* 3, 3 (2025), 1–27.

[26] Wensheng Luo, Yixiang Fang, Chunxu Lin, and Yingli Zhou. 2024. Efficient Parallel D-Core Decomposition at Scale. *Proc. VLDB Endow.* 17, 10 (2024), 2654–2667.

[27] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management (CIKM)*. 469–478.

[28] Amir Mehrafsa, Sean Chester, and Alex Thomo. 2020. Vectorising k-core decomposition for gpu acceleration. In *Proceedings of the 32nd international conference on scientific and statistical database management (SSDBM)*. 1–4.
[29] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2011. Distributed k-core decomposition. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on principles of distributed computing*. 207–208.
[30] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *ACM SIGGRAPH (SIGGRAPH '08)*. Article 16, 14 pages.
[31] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. White Paper.
[32] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.
[33] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. 2024. Guardian: Safe gpu sharing in multi-tenant environments. In *Proceedings of the 25th International Middleware Conference (Middleware)*. 313–326.
[34] Siani Pearson. 2009. Taking account of privacy when designing cloud computing services. In *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (ICSE Workshop)*. IEEE, 44–52.
[35] Christoph Schweimer, Christine Gfrerer, Florian Lugstein, David Pape, Jan A Velimsky, Robert Elsässer, and Bernhard C Geiger. 2022. Generating simple directed social network graphs for information spreading. In *Proceedings of the ACM web conference (WWW) 2022*. 1475–1485.
[36] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.
[37] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE, 469–478.
[38] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2018. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems* 54, 3 (2018), 677–710.
[39] Xibo Sun and Qiong Luo. 2023. Efficient gpu-accelerated subgraph matching. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 2 (2023), 1–26.
[40] John White. 2025. *What is the lead time for H200 delivery in 2025?* https://www.szwecent.com/what-is-the-lead-time-for-h200-delivery-in-2025/ Accessed: 2026-01-26.
[41] Feng Yanghe and Wang Jiao Teng. 2013. Peeling the drug target proteins network by a core decomposition method. *Research Journal of Biotechnology* 8, 10 (2013), 66–70.
[42] Chen Zhao, Ting Yu, Zhigao Zheng, Yuanyuan Zhu, Song Jin, Bo Du, and Dacheng Tao. 2024. PICO: Accelerating All k-Core Paradigms on GPU. In *Proceedings of the 8th Asia-Pacific Workshop on Networking (APNet)*. 221–222.
[43] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2018. Diversified coherent core search on multi-layer graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 701–712.

Received October 2025; revised January 2026; accepted February 2026