Bound-Tightened Densest Subgraph Discovery on GPU

WAJID MANZOOR, Rowan University, USA
KE FAN, Temple University, USA
MUHAMMAD SHAHEER, Rowan University, USA
GUIMU GUO, Rowan University, USA

The densest subgraph discovery (DSD) problem is a fundamental task in graph mining with applications in social networks, bioinformatics, graph databases, and systems engineering. Given a graph $G = (V, E)$ and an integer $k \geq 2$, the goal is to find a vertex subset $D \subseteq V$ whose induced subgraph $G(D)$ maximizes the k-clique density, defined as the number of k-cliques per vertex. Larger values of k capture higher-order connectivity patterns beyond edges, enabling the discovery of more cohesive structures. We present a GPU-accelerated framework that integrates a warp-level k-clique enumeration algorithm along with clique-core–based pruning and edge pruning techniques to reduce the search space, followed by parallel connected component decomposition to split the pruned graph into independent smaller subproblems. The exact solution is then obtained by formulating DSD on each connected component as a maximum flow problem and solving it using a highly parallel vertex-centric push–relabel algorithm, enhanced with tight upper and lower density bounds and compact, GPU-friendly data structures. Experiments on a diverse set of real-world and synthetic graphs show substantial speedups over the state-of-the-art CPU implementation, while producing identical solutions.

CCS Concepts: • **Mathematics of computing → Graph algorithms**.

Additional Key Words and Phrases: Densest Subgraph Discovery; Parallel Graph Mining; CUDA

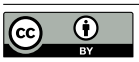
ACM Reference Format:
Wajid Manzoor, Ke Fan, Muhammad Shaheer, and Guimu Guo. 2026. Bound-Tightened Densest Subgraph Discovery on GPU. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 146 (June 2026), 25 pages. https://doi.org/10.1145/3802023

1 Introduction

Dense subgraph discovery (DSD) is a fundamental primitive in graph mining, enabling the identification of highly cohesive structures essential for many domains, such as community detection in social networks [4, 20], regulatory module discovery in bioinformatics [8, 16], efficient indexing in graph databases [5, 13], and cohesive subgraph extraction in visualization [21, 22].

Most previous work [7, 14] defines density in terms of edge count. For example, in Figure 1, if we use the edge density formulation ($k = 2$), the densest subgraph is D_1, with an edge density of $\frac{19}{10} = 1.9$. However, when we switch to triangle density ($k = 3$), the densest subgraph changes to D_2, with a density of $\frac{4}{4} = 1.0$. This illustrates the advantage of higher-order formulations: while D_1 contains more edges overall, D_2 forms a tightly-knit structure where every vertex participates in multiple triangles, revealing a stronger notion of cohesiveness than edge density alone [19]. The general k-clique formulation allows density to capture richer connectivity patterns, such as strong

Authors' Contact Information: Wajid Manzoor, Rowan University, Glassboro, NJ, USA, wajidm36@students.rowan.edu; Ke Fan, Temple University, Philadelphia, PA, USA, ke.fan@temple.edu; Muhammad Shaheer, Rowan University, Glassboro, NJ, USA, shahee22@rowan.edu; Guimu Guo, Rowan University, Glassboro, NJ, USA, guog@rowan.edu.



ACM 2836-6573/2026/6-ART146
https://doi.org/10.1145/3802023

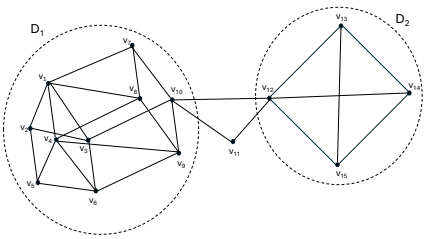


Fig. 1. A Toy Graph

Table 1. Summary of Notations Used in the Paper

Notation	Description
$G = (V, E)$	Undirected graph with vertices V and edges E
$G(D)$	Subgraph induced by vertex subset $D \subseteq V$
$N(v)$	Set of neighbors of vertex v in G
$d(v)$	Degree of vertex v in G, $\lvert N(v)\rvert$
(k, ψ)	Set of all k-clique instances in G
$d_G(v, \psi)$	k-clique degree of vertex v
$n(G, \psi)$	Total number of k-clique instances in G
$\rho(G, \psi)$	k-clique density of G
D^*	Subgraph of G with maximum k-clique density

local clustering with $k = 3$ or tight communities with $k > 3$, making it more informative for tasks like community detection [19], network analysis [17], and anomaly detection [1].

Existing exact methods for k-clique DSD [7, 9, 19] often formulate the problem as a sequence of maximum flow computations over clique-induced networks. While these approaches are effective for small to medium graphs and guarantee exact solution, they are typically implemented as CPU serial algorithms, which limits scalability to large datasets. Storing all k-cliques and the flow networks incurs high memory overhead, especially for dense graphs where the number of cliques grows rapidly with k, and the computational cost increases correspondingly. As k increases, the exponential growth in cliques further amplifies these scalability issues, making existing solutions impractical for many real-world scenarios.

Modern GPUs, with their massive parallelism and high memory bandwidth are ideal for accelerating DSD. However, exploiting GPUs also introduces four key challenges: **(1) Irregular memory access:** k-clique enumeration and flow updates involve scattered neighbor lookups that reduce memory coalescing. **(2) Recursive components:** Clique enumeration, clique-core decomposition, and connected component decomposition are inherently recursive, which is poorly suited to GPU execution. **(3) Memory constraints:** Storing all k-cliques and the induced flow network can quickly exceed available GPU memory in dense graphs. **(4) Workload imbalance:** Clique memberships are unevenly distributed across vertices, making it difficult to balance work across warps.

To address these challenges, we design a GPU-accelerated DSD framework that follows a *reduce–divide–solve* paradigm. **Reduce:** Warp-level k-clique enumeration with clique core based and edge pruning to eliminate low-density vertices and non-contributing edges. **Divide:** Parallel connected component decomposition to split the pruned graph into smaller, independent subproblems. **Solve:** Formulate DSD on each component as a maximum flow problem and solve it using a highly parallel push–relabel algorithm with novel tight bounds and GPU-friendly data structures. Despite pruning and parallelization, all steps preserve feasibility, and the final solution is guaranteed to be exact.

This design exploits fine-grained GPU parallelism while minimizing memory overhead and redundant computation. Our main contributions are:

- **GPU-optimized enumeration and pruning:** Warp-level k-clique enumeration integrated with clique-core based pruning and edge pruning to reduce the search space before exact computation.
- **Tighter component-specific upper bound:** A stronger, component-level upper bound on clique density that reduces the search space in the binary search for maximum density, lowering the number of required flow computations.
- **Flow network termination criteria:** A maximum height bound of $2|V_C|$ derived for our clique-induced flow networks, reducing unnecessary relabel operations.
- **Cheaper, structure-aware global relabeling:** A global relabel heuristic that exploits the regular structure of clique-induced networks to update distance labels more efficiently than generic approaches.

- **Flow network exact solver:** A vertex-centric push-relabel maximum flow algorithm adapted for clique-induced flow networks, integrating the above bounds and heuristics for faster convergence.
- **Performance gains:** Extensive experiments on real-world and synthetic graphs demonstrate up to 10^4 speedup over the fastest CPU baseline, while producing identical solutions.

2 Preliminaries

This section presents the essential background for our approach. We first define the graph model, k-clique density, and the densest subgraph problem in Section 2.1. We then review the maximum flow formulation used for exact feasibility testing in Section 2.2.

2.1 Definitions

Graph Notation. We consider an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. For any subset $D \subseteq V$, let $G(D)$ denote the subgraph induced by D, containing all edges in E whose endpoints both lie in D. Table 1 summarizes the key notations used throughout the paper.

k-Clique Instance Set. Given a graph $G = (V, E)$ and an integer $k \geq 2$, let (k, ψ) denote the set of all k-clique instances in G. Each k-clique instance is an unordered subset of vertices $S \subseteq V$ with $|S| = k$ such that every pair $u, v \in S$ is connected by an edge in E. Permutations of the same vertex set are considered the same instance. For example, in Figure 1, subgraph D_2 contains exactly one 4-clique instance.

Clique Degree. For a vertex $v \in V$, the *clique degree* with respect to (k, ψ), denoted $d_G(v, \psi)$, is the number of k-clique instances in (k, ψ) that contain v.

k-Clique Core and Clique Core Value. Given a $\tau \geq 0$, the τ *k-clique core* of G, denoted $C_k(\tau)$, is the maximal induced subgraph $G(D)$ wherein every vertex $v \in D$ satisfies $d_{G(D)}(v, \psi) \geq \tau$. $d_{G(D)}(v, \psi)$ counts the number of k-clique instances (from (k, ψ)) contained in the induced $G(D)$ that include v. The *k-clique core value* of a vertex v, denoted $\kappa_k(v)$, is the largest τ such that $v \in C_k(\tau)$. Clique-core decomposition computes $\kappa_k(\cdot)$ by iteratively removing vertices whose current k-clique degree falls below the threshold, analogous to classic k-core decomposition.

Component Decomposition. For a graph $G = (V, E)$, component decomposition partitions V into disjoint sets $V_1, V_2, \ldots, V_r$ such that each induced subgraph $H_i = G(V_i)$ is connected, and for any $i \neq j$ there is no edge in E_H with one endpoint in V_i and the other in V_j. We refer to $\{H_1, \ldots, H_r\}$ as the connected components of G

k-Clique Density. Let (k, ψ) be the set of all k-clique instances in G. The *k-clique density* of G is defined as:

$$\rho(G, \psi) = \frac{n(G, \psi)}{|V|},$$

where $n(G, \psi)$ is the total number of k-clique instances in G.

Problem Definition. Given a graph $G = (V, E)$ and the set (k, ψ) of all k-clique instances in G with $k \geq 2$, the goal is to find a vertex subset $D \subseteq V$ such that the induced subgraph $G(D)$ maximizes the k-clique density:

$$D^* = \arg\max_{D \subseteq V} \rho(G(D), \psi),$$

where $\rho(G(D), \psi) = \frac{n(G(D), \psi)}{|D|}$ and $n(G(D), \psi)$ denotes the number of k-clique instances in $G(D)$.

This formulation naturally admits an exact feasibility test via maximum flow, which forms the basis of both prior CPU algorithms and our GPU implementation.

2.2 Maxflow Problem

Unlike traditional mining tasks such as motif counting [2], DSD is a multi-stage optimization process that need clique enumeration and iterative flow-based refinement, searching over a candidate density value α [7, 19]. For a fixed α, a feasibility test determines whether an induced subgraph $G(D)$ exists with k-clique density $\rho(G(D), \psi) \geq \alpha$. This feasibility test can be reduced to a maximum flow problem on a clique-induced flow network that trades off (i) selecting vertices into D, and (ii) gaining credit for each k-clique instance fully contained in D. If the minimum cut has finite capacity below a threshold, then such a set D exists, otherwise, no subgraph can achieve density α. By repeating this test for different α values via binary search, we obtain an exact solution.

Since binary search requires numerous feasibility tests, runtime is dominated by repeated maximum flow computations. Edmonds-Karp [6] is a classic BFS-based maximum flow algorithm. But its repeated global BFS traversals provide limited parallelism and irregular memory access, hindering GPU efficiency. In contrast, push-relabel [10] algorithm relies on localized updates that expose substantial concurrency, aligning well with GPU throughput-oriented execution while preserving the exactness.

3 Related Work

DSD problem has been extensively studied under various density definitions, most commonly edge density [9, 13, 22]. Early approaches, such as Goldberg's exact algorithm [9] and Khuller and Saha's approximation schemes [13], focused on maximizing the average degree, which can be solved in polynomial time. Extensions to quasi-cliques and other cohesiveness measures have been proposed to better capture dense regions in real-world networks [4, 17, 20], with applications in community detection [8], biological networks [16], and social networks [22].

For richer connectivity patterns, Tsourakakis [19] introduced the exact k-clique densest subgraph formulation, where density is measured by the number of k-clique instances per vertex. This generalization allows tuning k to control the cohesiveness of detected subgraphs, with $k = 2$ reducing to edge density and higher k capturing stronger local connectivity. Recent work by Fang et al. [7] proposed an exact CPU algorithm for k-clique DSD, combining clique-core decomposition, pruning, and a maximum flow–based feasibility check. While effective, its sequential nature, recursive components, and memory-intensive data structures limit scalability.

Alternative formulations for DSD have been explored using linear programming (LP) [9], integer programming (IP) [16], and branch-and-bound search [20]. While LP and IP offer exact solutions, they require solving large-scale optimization problems over dense constraint matrices, making them computationally expensive and memory-intensive for large graphs.

Despite the inherent parallelism in clique enumeration, pruning, and component decomposition, few studies have explored GPU acceleration for DSD. Existing GPU research has largely focused on subgraph enumeration [3], connected component decomposition [18], and generic maximum flow solvers [12]. However, to the best of our knowledge, no prior work provides a complete, exact GPU implementation for k-clique densest subgraph discovery. Our study fills this gap by designing a GPU-optimized, flow-based framework that incorporates tighter bounds, structure-aware heuristics, and memory-efficient data structures, enabling significant speedups over state-of-the-art algorithms.

Baseline Approach on CPU: The CPU baseline algorithm [7] computes an exact solution to the k-clique densest subgraph discovery problem. It combine safe pruning, component-wise decomposition, and a sequence of feasibility tests on motif-induced flow networks. It first applies clique-core pruning to retain all vertices that can appear in an optimal solution, and then decomposes the remaining graph into independent connected components. For each component, lower and upper bounds on the optimal density are derived from clique-core properties to define a valid search

Algorithm 1 Simplified DSD Algorithm (adapted from [7])

Require: Graph $G = (V, E)$, clique size k
Ensure: Densest subgraph D^*

1. Perform k-clique core decomposition of G
2. Let (k^*, ψ) be the densest clique-core
3. Remove every edge that is not part of any k-clique
4. $C \leftarrow$ connected components of the (k^*, ψ)-core
5. Set lower (l) and upper (u) bounds on k-clique density
6. **for all** $C = (V_C, E_C) \in C$ **do** ▷ Process each component
7. **while** $u - l \geq \frac{1}{|V_C|(|V_C|-1)}$ **do**
8. $\alpha \leftarrow \frac{l+u}{2}$
9. Build flow network from C with threshold α
10. Compute a minimum s–t cut (S, T)
11. **if** $S = \{s\}$ **then** ▷ Only source in source side
12. $u \leftarrow \alpha$
13. **else**
14. $U \leftarrow S \setminus \{s\}$ ▷ Vertices on source side
15. $l \leftarrow \alpha$
16. **if** $\rho(G[U], \psi) \geq \rho(D^*, \psi)$ **then**
17. $D^* \leftarrow G[U]$ ▷ Update best solution

interval. Within this interval, it performs a binary search over candidate density values α. For each α, a flow network is constructed whose minimum s–t cut determines whether the component contains a subgraph with k-clique density at least α. Since feasibility is monotonic in α and the bounds safely enclose the optimum, the binary search converges to the maximum feasible density, and the resulting vertex set yields the exact densest subgraph.

The CPU baseline maintains a valid search interval for the optimal k-clique density using safe lower and upper bounds. The lower bound is initialized as the maximum k-clique density observed so far, among the densest k-clique core and connected components. The upper bound is given by the maximum k-clique core value within the connected component. Intuitively, this value represents the largest possible average k-clique participation any vertex subset in the component can achieve. No induced subgraph can have k-clique density exceeding this bound, making it a safe upper limit.

Algorithm: As outlined in Algorithm 1, the algorithm begins by performing clique-core decomposition to identify the densest clique-core (k^*, ψ) (lines 1–2). Vertices whose k-clique core values fall below the density of this core are removed, as their inclusion would reduce overall density. All edges that do not participate in any k-clique are then eliminated (line 3), and the remaining graph is partitioned into connected components (line 4).

For each component, the algorithm sets the lower bound l to the maximum k-clique density among all components, and the upper bound u to the maximum k-clique core value (line 5). A binary search is then conducted over the interval $[l, u]$ (lines 6–7). In each iteration, the algorithm constructs a flow network encoding the current density threshold α (line 9) and computes a minimum s–t cut (line 10) to assess feasibility. If the cut yields only the source vertex on the source side, the upper bound is decreased (lines 11–12). Otherwise, the set of vertices on the source side is considered a valid subgraph, and the lower bound is increased (lines 13–15). If this subgraph achieves a density higher than the best found so far, it becomes the current best solution (lines 14–15). After processing all components, the densest valid subgraph is returned as the final result (lines 16–17).

Example: We illustrate the CPU baseline on Figure 1 with triangles ($k = 3$). The graph contains 11 triangles. Clique-core decomposition assigns $\kappa_3(v) = 1$ for vertices $\{v_1, \ldots, v_{11}\}$ and $\kappa_3(v) = 3$ for $\{v_{12}, \ldots, v_{15}\}$. clique-core exhibits triangle density 1, establishing the global lower bound to $l = 1$ with no vertex removed. After discarding edges not participating in triangles (e.g., (v_2, v_5), (v_3, v_6), ...), the graph splits into two components induced by $\{v_1, \ldots, v_9\}$ and $\{v_{10}, \ldots, v_{15}\}$. For each component C, the algorithm sets $u_C = \max_{v \in V_C} \kappa_3(v)$, giving $u_C \in \{1, 3\}$, and binary searches $\alpha \in [l, u_C]$ via the flow based feasibility test yielding the best solution D_2. The densest clique-core and the final densest subgraph coincide here, but they need not be identical in general.

While this method guarantees exact solution and incorporates effective pruning with flow-based verification, its sequential design limits scalability. Moreover, the data structures used for storing k-cliques and constructing flow networks incur high memory overhead and require frequent backward-edge lookups in the residual graph, increasing both space and computational costs. These limitations motivate the development of a parallel formulation capable of leveraging modern hardware accelerators, such as GPUs, to scale efficiently to large graphs.

4 Novel Density Bounds

In this section, we propose two novel tightened upper bounds for DSD algorithm. Prior work has proposed various strategies to derive lower and upper bounds on the optimal clique density $\rho(D^*)$, often relying on core values or worst-case structural assumptions. Let $G(V, E)$ be a graph containing a total of t k-cliques. Let (k^*, ψ) denote the densest k-clique core, and let $k'_{\max}$ be the maximum clique-core value in G. We denote the optimal density of the densest subgraph D^* as $\rho(D^*)$.

4.1 Lower Bounds

Tsourakakis et al. [19] provide a trivial lower bound:

$$\frac{t_\psi}{|V_\psi|} \leq \rho(D^*),$$

where t_ψ is k-cliques count in the core and $|V_\psi|$ is its vertex count.

Fang et al. [7] propose a tighter bound by noting that each vertex in the densest k-clique core participates in at least k^* cliques. Since each clique includes k vertices, the number of cliques is at least $\frac{k^* \cdot |V_\psi|}{k}$, leading to:

$$\frac{k^*}{k} \leq \rho(D^*).$$

We adopt this bound but compute it across all connected components, selecting the maximum. This results in a tighter, more localized lower bound.

4.2 Upper Bounds

Two common strategies exist for upper bounding $\rho(D^*)$:

(1) Worst-Case (Complete Graph) Bound from [19]:

$$\rho(D^*) \leq \frac{\binom{n}{k}}{n} = \frac{(n-1)(n-2)\ldots(n-k+1)}{k!},$$

based on the observation that a complete graph maximizes the number of k-cliques for any given n, where n is the number of vertices.

(2) Structural Bound from Fang et al. [7]: $\rho(D^*) \leq k_{\max}$, where $k_{\max}$ is the maximum clique-core value.

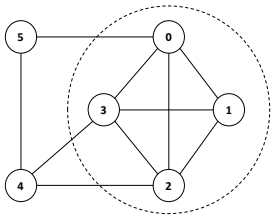


Fig. 2. Density Bounds.

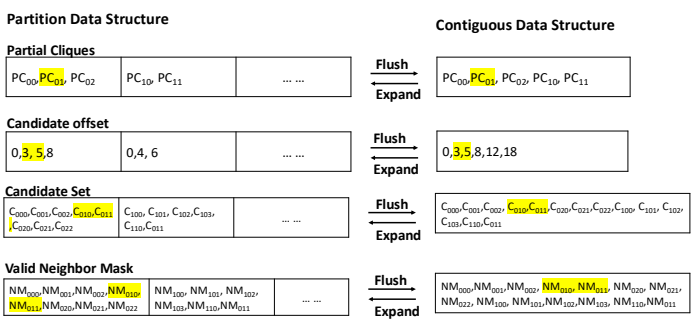


Fig. 3. Clique Enumeration Intermediate Data Structure. *Expand:* Warp retrieves a partial clique from contiguous arrays, expands it, and stores results in its assigned partition. *Flush:* After expansion, each warp writes partial cliques from its partition back to contiguous arrays for the next stage.

Proposed Upper Bound I: We introduce a new, data-aware upper bound based on the minimum number of vertices x required to accommodate t k-cliques. Let x be the smallest integer such that:

$$\binom{x}{k} \geq t.$$

This formulation avoids the overly loose assumption of a complete graph and captures the minimal subgraph size needed to host the given number of cliques.

We get :

$$\binom{x}{k} = \frac{x(x-1)\cdots(x-k+1)}{k!}.$$

Which yields:

$$\rho(D^*) \leq \frac{t}{x} \leq \frac{\binom{x}{k}}{x} = \frac{(x-1)(x-2)\ldots(x-k+1)}{k!}.$$

We denote this bound as

$$UB_I = \frac{(x-1)(x-2)\ldots(x-k+1)}{k!}$$

where x is the smallest integer with $\binom{x}{k} \geq t$.

$x \leq n$ makes it always tighter than the complete-graph bound:

$$\frac{(x-1)(x-2)\ldots(x-k+1)}{k!} \leq \frac{(n-1)(n-2)\ldots(n-k+1)}{k!}.$$

In certain cases, our proposed bound outperforms the $k_{\max}$-based estimate. For example, in the graph shown in Figure 2 with $k = 3$, the true densest subgraph is $\{0, 1, 2, 3\}$, containing $t = 5$ cliques with density $\rho(D^*) = 1$. Here, $k_{\max} = 3$, while our bound gives $x = 5$, leading to:

$$\rho(D^*) \leq \frac{5}{5} = 1,$$

which exactly matches the optimal value.

To further tighten this estimate, we compute the bound *component-wise* rather than globally. Since the number of cliques t is typically smaller within each connected component, the resulting value of x is also smaller, producing a tighter bound.

Asymptotic Scaling: We also analyze the asymptotic scaling of this bound. For $t \gg k$, the inequality $\binom{x}{k} \geq t$ becomes:

$$x(x-1)\cdots(x-k+1) \geq k! \cdot t.$$

Approximating the left-hand side as x^k, we get:

$$x^k > k! \cdot t \quad \Rightarrow \quad x \propto t^{1/k}.$$

Hence, the upper bound scales as:

$$\frac{t}{x} \propto t^{\frac{k-1}{k}}.$$

This implies that the bound grows *sublinearly* with t, since $\frac{k-1}{k} < 1$, meaning smaller components with lower t naturally yield tighter estimates on clique density.

Proposed Upper Bound II: We further propose an *entropy-based* upper bound that leverages the distribution of k-clique participation across vertices. Let

$$p_v = \frac{d(v,\psi)}{k \cdot t}$$

be the normalized participation probability of vertex v, where $d(v,\psi)$ is its k-clique degree and t is the total number of k-cliques. By definition,

$$\sum_{v \in V} p_v = 1.$$

Let s' denote the minimum number of vertices required to contain all t k-cliques. Clearly, the maximum k-clique density cannot exceed the case where all t cliques are concentrated in these s' vertices:

$$\rho(D^*) \leq \frac{t}{s'}.$$

Directly computing s' is equivalent to solving a combinatorial set-cover problem, which is intractable for large graphs. Instead, we exploit a result from information theory: the *support size* of any probability distribution $\{p_v\}$ is bounded below by the inverse of its second moment, also known as the Rényi entropy of order 2 [15]:

$$s' \geq \frac{1}{\sum_v p_v^2}.$$

This inequality follows Cauchy–Schwarz inequality [11]:

$$\left(\sum_v p_v\right)^2 \leq s' \cdot \sum_v p_v^2.$$

Combining these results yields:

$$\rho(D^*) \leq \frac{t}{s'} \leq t \cdot \sum_v p_v^2.$$

We denote this bound as

$$UB_{II} = t \sum_v p_v^2, \quad p_v = \frac{d(v,\psi)}{kt}.$$

Recall that our combinatorial bound is

$$UB_I = \frac{t}{x}, \quad \text{where } x \text{ is the smallest integer with } \binom{x}{k} \geq t.$$

UB_{II} is tighter when $\sum_v p_v^2 < \frac{1}{x}$, that is when clique participation is relatively uniform across many vertices (high Rényi entropy). Conversely, UB_I is tighter when participation is concentrated

among a few vertices. Since they capture distinct structural aspects, combinatorial vs. distributional, neither bound universally dominates the other. Relative to the clique-core bound $k_{\max}$, note:

$$UB_{II} = \frac{1}{k^2 t} \sum_v d(v, \psi)^2 \leq \frac{d_{\max}}{k},$$

where $d_{\max}$ is the maximum clique degree. Thus, UB_{II} is guaranteed to be tighter than $k_{\max}$ when $d_{\max} < k \cdot k_{\max}$. In real-world graphs, UB_{II} often outperforms $k_{\max}$, as shown in Section 7, though no universal dominance exists. In practice, we take the minimum of all three upper bounds to maximize tightness.

We compute both bounds component wise but exploit different granularities of parallelism. Let the graph decompose into N connected components, each of average size c, and let T denote the number of threads. For UB_I, we parallelize across components, each component requires finding the smallest x such that $\binom{x}{k} \geq t$, where t is the number of k-cliques in that component. Since $x \leq c$, a linear scan over $x \in [k, c]$ costs $O(c)$ time per component, and assigning one component per thread yields a parallel time of $O(Nc/T)$. In contrast, UB_{II} is parallelized within a component, we compute $p_v = d(v, \psi)/(kt)$ and evaluate $\sum_v p_v^2$, which is a reduction over the component's c vertices and takes $O(c/T)$ time per component.

The derived bounds initialize the search interval for the flow based feasibility test. Feasibility is monotonic in the density threshold α, if no subgraph satisfies α, then none exists for any larger value. Thus, tighter bounds reduce the number of binary search iterations without affecting correctness.

5 Our System Design

Our system implements an end-to-end, exact GPU pipeline for k-clique densest subgraph discovery. Starting from the input graph, we enumerate all k-cliques on the GPU using a warp-centric strategy. We then perform clique-core decomposition and prune edges that do not participate in any clique. The reduced graph is decomposed into connected components, each of which is processed independently. For each component, we construct a clique induced flow network and solve a sequence of feasibility tests using a GPU-optimized push relabel algorithm guided by tight bounds. The densest solution across all components is returned as the exact result.

5.1 Overview and Challenges

Task-based design. To list all k-cliques and perform clique-core decomposition efficiently, we adopt a task-based parallel framework. The input is partitioned into independent tasks, each assigned to a separate warp. This approach maximizes parallelism, minimizes memory contention, and allows for efficient execution by ensuring that each warp operates independently on its assigned subproblem.

Vertex-centric approach. One key challenge in designing an efficient maximum flow solver for clique-induced networks lies in reducing unnecessary computation during flow updates. Traditional edge-centric push–relabel algorithms scan all edges in each iteration, regardless of whether the source vertex has excess flow or satisfies the height condition. This results in substantial redundant work and poor parallel efficiency, especially on GPUs. To address this, we adopt a vertex-centric push–relabel approach, where only active vertices—those with positive excess—are processed. This significantly reduces the search space and improves workload balance. Recent work by Hsieh et al. [12] demonstrates that vertex-centric designs can achieve up to 7× speedup over edge-centric counterparts by avoiding edge-level scanning and thread divergence on GPUs. Warps process active vertices iteratively, using cooperative groups for efficient synchronization and atomic operations. This approach improves memory locality, reduces synchronization overhead, and scales well on GPUs, particularly for sparse graphs.

Choice of computational unit. Our system contains multiple computationally intensive components, k-clique enumeration, clique core decomposition, and the dynamic exact solver, each with different performance characteristics. Selecting an efficient GPU computational unit was therefore a key design decision. We choose the warp as our fundamental execution unit, as it offers an optimal balance between fine-grained parallelism, coalesced memory access, and low synchronization costs. Warps enable aligned memory access for neighbor lists, efficient reductions via warp level primitives such as `__shfl_down_sync`, and reduced control divergence, making them well-suited for the irregular workloads.

Memory representation challenges. Another major challenge is storing and accessing all k-cliques and the associated flow network over both vertices and cliques. The baseline CPU approach maintains two data structures: (1) an adjacency list where each vertex stores the k-cliques it participates in, and (2) a hash map from each clique to its constituent vertices. This supports two efficient lookups, retrieving all cliques containing a vertex in $O(d(v))$ time, and retrieving all vertices in a clique in $O(k)$ time, but it is highly memory-intensive. Each clique is stored k times in the adjacency list and once in the global map. On dense graphs with many cliques and limited GPU memory, this redundancy becomes a critical bottleneck.

For instance, in a fully connected graph with V vertices, the number of k-cliques is approximately

$$t = \binom{V}{k} \approx V^k \quad (\text{for } k \ll V).$$

The memory usage is $O(t \cdot k^2)$ for the adjacency list and $O(t \cdot k)$ for the global map. Even for moderate values of V and k, t can grow astronomically, making the total footprint infeasible for GPU execution. This motivates the need for a more compact and parallel-friendly representation, which we introduce in later sections.

On the CPU, the flow network is implemented as an adjacency list of hash maps, where each vertex stores outgoing neighbors with `(residual, capacity)` pairs. For every edge (u, v), both forward and backward edges are explicitly stored to enable constant-time residual updates. When flow is pushed along a forward edge, the corresponding backward edge must be updated accordingly. Using hash maps enables average-case $O(1)$ lookup time, supporting efficient flow adjustments. Overall, the CPU memory usage is $O(|V| + k + k \cdot t)$. While effective on CPUs, directly porting this design to GPUs is problematic: CUDA lacks native dynamic hash maps, hash lookups cause non-coalesced memory access, the irregular control flow increases warp divergence, concurrent updates require costly synchronization, and collisions add further overhead. These factors make hash maps unsuitable for GPU execution, motivating the compact, structure-aware GPU-friendly representations described in sections 5.2.

CPU vs. GPU Pipeline Comparison. While our GPU algorithm follows the same high-level framework as the CPU baseline, clique enumeration, clique core decomposition, component decomposition, and flow-based feasibility testing, each stage is fundamentally restructured for GPU execution rather than directly ported from the CPU implementation. Clique enumeration and clique-core decomposition adopt flat, array-based layouts and warp centric-task partitioning to maximize memory coalescing. It avoids pointer chasing and control divergence inherent in recursive CPU traversals, which are particularly inefficient on GPUs. Connected component decomposition uses label propagation to expose fine-grained parallelism across vertices. This design avoids the synchronization overhead of frontier-based BFS and improves SIMT scalability. For the flow feasibility test, we replace Edmonds–Karp with GPU-optimized push-relabel, enabling localized updates, higher arithmetic intensity, and better thread utilization. Additional optimizations, such as revised height rules, early termination, and structure-aware global relabeling, further reduce

redundant work. These design choices reflect GPU-aware reformulations tailored to parallelism, synchronization, and memory access, rather than a straightforward CPU-to-GPU port.

5.2 GPU-Aware Data Structures

Efficient data structures are critical for ensuring the scalability and performance of our DSD algorithm. We adopt compact, GPU-friendly representations for graphs, clique sets, and flow networks, enabling fast access and minimal memory overhead on memory-constrained devices.

Graph and Component Representation. The input graph, the intermediate Directed Acyclic Graph (DAG), and the connected components are all stored in Compressed Sparse Row (CSR) format. This includes an offset array (e.g., neighbor offset or component offset) and a contiguous neighbor list. This layout offers efficient traversal and indexing while ensuring coalesced memory accesses, making it well-suited to GPU execution.

Clique Representation. Storing all k-cliques is the most memory-intensive part of DSD. To reduce this cost, we use a flat, compact layout virtually partitioned into k segments, where each segment stores the i^{th} vertex of every k-clique at a fixed offset. For example, with 9 cliques and $k = 3$, we allocate a continuous array of 27 entries. The first 9 positions (segment 1) store the first vertex of each clique, the next 9 (segment 2) store the second vertex, and the final 9 (segment 3) store the third. Thus, for a clique $C = v_1, v_2, \ldots, v_k$, the v_i resides in the i^{th} segment, yielding an aligned and cache-friendly layout. We also maintain a status array to track each clique's membership in clique-core decomposition and connected components. This reduces the footprint to $O(t \cdot k)$, a significant improvement over CPU designs.

Although lookup operations in this layout are not linear-time as in CPU hash-based designs, our GPU implementation offsets this cost through massive parallelism. Searches are distributed across warps, blocks, or the full grid, effectively amortizing the lookup overhead and making the access pattern efficient in practice. The flat array-based clique representation is conceptually portable to a CPU implementation, but it is designed to exploit GPU-specific properties such as coalesced global memory access and warp-wide batched exploration. On CPUs, this layout would require linear scans over fixed-size arrays and does not benefit from pointer localized traversal or cache driven execution, making it significantly slower than hash-based adjacency structures.

During clique enumeration intermediate stages, the algorithm needs to store (1) *Partial Cliques:* subset of vertices of size ($r < k$) that are all mutually adjacent, (2) *Candidates:* vertices adjacent to every vertex in the partial clique, representing a possible extension to eventually form a k-clique and (3) *Valid Neighbors:* denotes the subset of candidates neighbors that intersect with current candidate set. To store these efficiently on GPU, we maintain two set of auxiliary arrays.

- Arrays for the number of partial cliques and the partial cliques themselves,
- Candidate offset and candidate lists arrays
- Bitmask arrays representing valid neighbors within each vertex's neighbor list.

Since *Partial Cliques* can reach maximum length $k - 2$ before final expansion, they can be stored without requiring offset array. *Valid Neighbor* sets can be large in dense graphs, so we efficiently encode them using bitmasks, which significantly reduces memory usage while enabling fast membership checks.

Both auxiliary sets share identical array types but differ in layout and purpose as shown in Figure 3. The first set is virtually partitioned across GPU warps, allowing each warp to write on its segment, enabling concurrent writes. It stores the partial cliques, candidate lists, and valid neighbor masks generated during the current enumeration depth. The second set uses a contiguous layout, where completed partial cliques are flushed at the end of the stage. These flushed arrays then serve

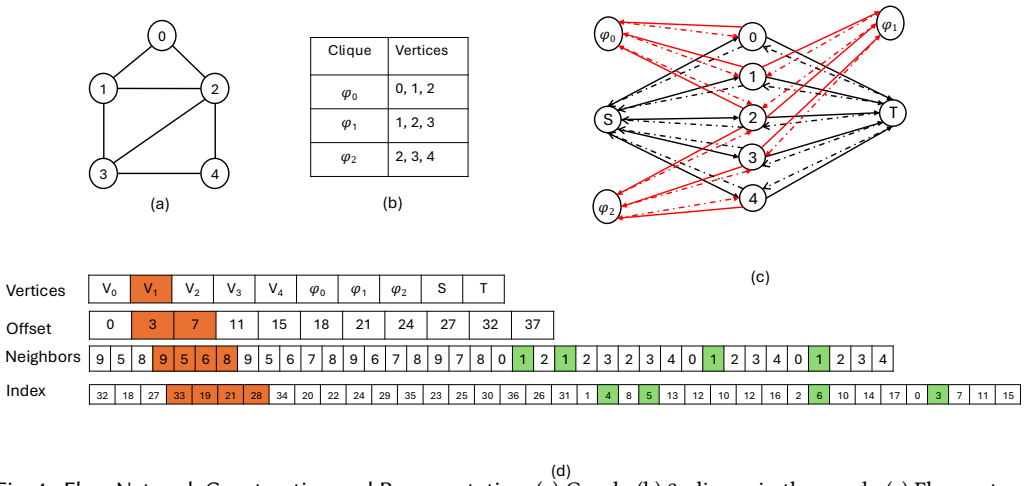


Fig. 4. Flow Network Construction and Representation. (a) Graph. (b) 3-cliques in the graph. (c) Flow network constructed using vertices and cliques: solid lines denote forward edges, dashed lines represent backward edges. Red edges connect vertices to cliques; black edges connect the source/sink to vertices. (d) Our CSR-based data structure: for vertex v_1, outgoing neighbors are highlighted in orange, and the corresponding backward edge indices are highlighted in green.

as the input for the next enumeration level, ensuring balanced workload distribution and efficient transition between stages.

Flow Network Representation. The flow network is also stored in CSR format for both forward and backward edges, using standard offset and neighbor arrays. Figure 4 illustrates this structure. To support constant-time ($O(1)$) reverse edge lookups, we add an index mapping array linking each forward edge to its corresponding backward edge and vice versa. This eliminates the need for hash maps, removing collision handling, synchronization costs, and non-coalesced access, while preserving the CPU implementation's asymptotic memory usage $O(|V| + t + k \cdot t)$.

6 Algorithm Design

6.1 Host Algorithm

The host program orchestrates the full GPU-accelerated DSD pipeline, encompassing graph preprocessing, k-clique enumeration, clique-core decomposition, flow network construction, and iterative refinement. Its objective is to transform the input graph $G = (V, E)$ into a flow network that encodes vertex–clique relationships, enabling the exact computation of the densest subgraph with respect to k-cliques. It manages kernel execution and oversees the end-to-end control flow, as outlined in Algorithm 2.

Algorithm: In line 1, the input graph is loaded in the CSR format, and the k-clique core numbers $\text{core}(v)$ are computed for all $v \in V$. A Directed Acyclic Graph (DAG) is then constructed (line 2), where an arc (v, u) exists only if $\text{core}(v) > \text{core}(u)$. This ordering induces a canonical search space: every k-clique $v_1, v_2, \ldots, v_k$ appears exactly once according to the core-value ordering, eliminating redundant enumerations caused by vertex permutations.

Next, in line 3, all k-cliques are enumerated on the GPU using a warp-centric modification of the Bron–Kerbosch algorithm [3]. We then compute the k-clique degree for each vertex, which serves as input to the clique-core decomposition step in line 4. Here, a peeling algorithm adapted for cliques is used to assign each vertex and clique a core value. This is followed by identifying the densest clique-core (k^*, ψ) (line 5). To reduce the problem size, we prune all vertices with clique-core values below the density of the densest core, and discard cliques that do not belong to

Algorithm 2 Host Program

Input: Graph $G = (V, E)$, clique size k
Output: Densest subgraph D^*

1. Load G in CSR format; compute core(v) for all $v \in V$
2. Construct DAG $G^{\rightarrow}$ with arcs (v, u) if core$(v) >$ core(u)
3. Enumerate all k-cliques $\mathcal{Q}$ in $G^{\rightarrow}$
4. Perform clique core decomposition
5. Identify densest clique-core (k^*, ψ)
6. Remove edges not involved in any clique
7. Decompose (k^*, ψ) into connected components $\mathcal{C} = \{C_1, \ldots, C_m\}$
8. Set lower (l), upper (u) bounds on k-clique density and $\alpha = (l + u)/2$
9. **for** each $C_i = (V_i, E_i) \in \mathcal{C}$ **do**
10. Build flow network for component C_i
11. Initialize `totalFlow` to 0
12. Send pre-flow from source to all vertices
13. **while** $(u - l) > \frac{1}{|V_i|(|V_i|-1)}$ **do**
14. **while** `sourceExcess + sinkExcess < totalFlow` **do**
15. Run push-relabel algorithm
16. Run global relabeling
17. Update flow network and extract $G(D)$
18. **if** $\rho(G(D), \psi) \geq \rho(D^*, \psi)$ **then**
19. $D^* \leftarrow G(D)$ ▷ Update best solution

the resulting subgraph. In line 6, we further prune all edges that are not part of any clique, which significantly reduces neighbor list sizes.

The reduced graph is decomposed into connected components $\mathcal{C} = \{C_1, C_2, \ldots\}$ using the Label Propagation Algorithm [18] (line 7), which offers better GPU scalability than BFS due to reduced synchronization and memory demands. Each clique is also assigned to the component containing its vertices.

In line 8, we initialize lower (l) and upper (u) bounds on clique density (see Section 4), and compute an initial guess $\alpha = (l + u)/2$. For each connected component (lines 9–20), we build a flow network over its vertices and corresponding cliques. We pre-fill the network by sending flow from the source to all vertices (line 11).

While the current bounds are not sufficiently tight, we solve the flow network using a push-relabel algorithm augmented with our custom global relabeling routine (lines 13–17). This relabeling method avoids BFS to better suit the GPU execution model.

After each iteration, we update the lower or upper bound, compute the new densest subgraph $G(D)$, and reconfigure the flow network with the updated threshold α (line 17). If the clique density of $G(D)$ exceeds that of the current best solution D^*, we update D^* accordingly (lines 18–19).

6.2 Clique Enumeration

This module enumerates all k-cliques using a level-wise expansion strategy (Algorithm 3). At each intermediate depth $r < k$, partial cliques and their associated metadata are stored using two logically distinct but structurally identical auxiliary layouts, as illustrated in Figure 3. At each level, warps retrieve partial cliques from a contiguous array generated in the previous iteration and write newly generated partial cliques, candidate sets, and valid neighbor masks into disjoint segments of

a partitioned auxiliary array. Because these write segments are non-overlapping by construction, concurrent writes are conflict-free and require no synchronization.

While partitioned storage ensures safe parallel generation, it leads to load imbalance due to the highly irregular number of partial cliques produced by different warps. To prevent imbalance from accumulating across levels, the *Flush* operation compacts all partitioned results into a single contiguous layout, which serves as the input for the next expansion stage. This alternating *Expand–Flush* design avoids recursion, pointer chasing, and dynamic memory allocation, while aligning clique enumeration with the SIMT execution model through regular memory access patterns and balanced workload distribution.

Algorithm: In algorithm 3, line 1 initializes these arrays. A bit-level label array is also initialized (Line 2) to help avoid duplicate candidates and to mark valid neighbors. Each bit corresponds to a vertex, and each warp is allocated a private $|V|$-bit label space to ensure independent processing. In the `listInitialCliques` kernel (Line 3), each warp is assigned a vertex (Line 14), and lanes iterate over its neighbors. If a neighbor's bit is not set, it is marked as a candidate and the bit is set using an atomic operation (Lines 15–20) to prevent write conflicts. If at least one candidate is found, the vertex is stored in the partial cliques array (Lines 20–21).

Next, each candidate's neighbors are scanned to identify those that are also marked in the label, producing a bitmask that encodes valid neighbors (Lines 21–27). This compact representation reduces memory usage compared to storing neighbor lists. The underlying idea is that a valid clique requires every candidate's neighbors to overlap with the current partial clique. After this, the label bits are cleared (Line 28), and the warp proceeds to the next vertex.

Lines 6–7 repeatedly invoke the two kernels `ListMidCliques` and `FlushPartition` until k' is reduced to 2. The `ListMidCliques` kernel functions similarly to `listInitialCliques`, except that it processes partial cliques from the contiguous arrays and attempts to extend them one candidate at a time by verifying clique validity. The `FlushPartition` kernel copies these intermediate cliques and their metadata to contiguous memory for the next round of expansion.

In Line 11, the `WriteFinalCliques` kernel processes each partial clique, selects a candidate, and combines it with its valid neighbors to generate complete k-cliques. The cliques are then stored in the final clique data structure using `atomicAdd` to determine the correct write location. The kernel also updates the clique degree of each participating vertex.

This design guarantees that every k-clique is enumerated exactly once, following the standard Bron-Kerbosch expansion with canonical ordering. Memory efficiency is achieved through bitmask based neighbor encoding. Conflict free concurrent writes are ensured by assigning each warp a disjoint partition of the output space during expansion. Load imbalance across enumeration levels is mitigated by contiguous compaction via the *Flush* operation, which redistributes work uniformly for subsequent stages. Together, canonical ordering, partitioned concurrent writes, and Expand-Flush compaction constitute key GPU oriented design choices that enable scalable clique enumeration. Section 7.6 empirically validates their impact through ablation studies that selectively disable canonical ordering, warp local compaction and compare against the full configuration.

6.3 Clique Core Decomposition

Clique-core decomposition serves pruning rule that removes vertices incapable of belonging to an optimal solution. It substantially reduces the size of the graph and clique set passed to downstream stages, essential for scalability on GPUs, where memory footprint dominate performance.

Lemma 1 (Clique-Core Pruning) Let (k^*, ψ) denote the densest k-clique core of G, and let $\rho_{\text{core}} = \rho(G(C_k(k^*)), \psi)$ be its k-clique density. Any vertex v with clique core value $\kappa_k(v) < \rho_{\text{core}}$ cannot belong to an optimal densest subgraph D^*.

Algorithm 3 Clique enumeration

Input: Graph $G = (V, E)$, clique size k, p_{size}
Output: All k cliques, clique degree

1. Allocate partitioned and contiguous array sets for partial cliques
2. Allocate warp-private label array $L \in \{0,1\}^{|V| \times \text{total warps}}$
3. Launch `List Initial Cliques` kernel
4. Flush results to contiguous arrays via `Flush Partition` kernel
5. $k' = k - 1; level = 1$
6. **while** $k' > 2$ **do**
7. Launch `List Mid Cliques` on contiguous partial cliques
8. Flush results to contiguous arrays via `Flush Partition` kernel
9. $k' \leftarrow k' - 1, level \leftarrow level + 1$
10. `Count Cliques` Kernel
11. Launch `Write Final Cliques` to write all k-cliques.

List Initial Cliques Kernel

12. **For each warp:**
13. Initialize shared counter to 0
14. $v \leftarrow$ warp ID (assigned vertex)
15. **for** `each lane in the warp` **do**
16. $u \leftarrow$ neighbor of v
17. `mask` $\leftarrow 1 \ll (u \bmod 32)$, `word`$_{pos} \leftarrow \lfloor u/32 \rfloor$
18. `old` $\leftarrow$ `atomicOr(&Label[word`$_{pos}$`], mask)`
19. **if** $\neg$(`old` & `mask`) **then**
20. Store u in candidate array at `atomicAdd(counter, 1)`
21. **if** `counter` > 0 **then**
22. Store v in partial clique array
23. **for** `each lane in the warp` **do**
24. $u \leftarrow$ corresponding candidate
25. **for** `each 32-neighbor chunk of` u **do**
26. Construct bitmask of labeled neighbors
27. Store bitmask in validNeighMask array
28. Clear all label bits using `atomicAnd`

Proof Sketch. By definition, every vertex $v \in D^*$ must participate in at least $\rho(D^*)$ k-cliques in the induced subgraph $G(D^*)$. Since the densest clique core achieves density ρ_{core}, we have $\rho(D^*) \geq \rho_{\text{core}}$. Consequently, all vertices of D^* must have $\kappa_k(v) \geq \rho_{\text{core}}$.

To compute clique-core values, we design an iterative, peeling-inspired algorithm tailored to higher-order structures (see algorithm 4). At each level, vertices whose clique core value equals the current level are removed. In contrast to classical core decomposition, which propagates updates through edges, our method updates all k-cliques incident to a removed vertex, decrementing the clique core values of the remaining vertices in those cliques. This captures clique-level connectivity instead of pairwise adjacency. The algorithm is parallelized using a warp centric batching strategy. Vertices selected at each level are processed concurrently, with each warp handling one removed vertex and scanning its incident cliques across lanes. Clique core updates are performed via atomic decrements on shared counters, ensuring correctness under concurrent updates while avoiding global synchronization.

Algorithm: In algorithm 4, lines 1–2 allocate two arrays, storing the selected vertices for removal (`buffer`) and the per-block counts (`bufferTails`). Line 3 initializes `count` (the total number of removed vertices) and `level` (the current clique-core level). Line 4 initialized the clique-core values as the vertices' clique degrees.

The main loop (Lines 5–7) continues until all vertices are peeled. In each iteration, the `SelectNodes` kernel scans each block's portion of vertices and adds those with clique-core value equal to the current level into the `buffer`. The corresponding `bufferTail` tracks how many such vertices are found per block.

Lines 8–10 initialize per-block variables: `bufTail` records the total number of vertices to process, and `base` tracks the starting index in the buffer. In Line 12, each warp processes one vertex. If its assigned index exceeds `bufTail`, the warp exits. Lines 15–16 increment the base index by the number of warps per block, allowing the block to iterate over its share of the buffer.

In Line 17, a vertex is read from the buffer. Each lane then scans the cliques in chunks (Lines 18–19) to check whether the current clique contains this vertex. If so, the lane iterates over all members of the clique (Lines 21–25), decrementing their clique-core values. If a neighbor's updated core value matches the current level, it is added to the buffer for the next iteration. Finally, the clique's status is updated in Line 26 to mark it as processed at the current level.

Clique-core pruning substantially reduces the number of vertices and k-cliques processed downstream, shrinking flow networks and accelerating the exact solver.

6.4 Component Decomposition

Our approach solves the densest subgraph discovery problem through a sequence of flow-based feasibility tests. To reduce the size and cost of these tests, we decompose the pruned graph into connected components and process each component independently. Since k-cliques cannot span connected components, densest subgraph must be fully contained within a single component. Therefore, component decomposition preserves exactness while enabling substantial reductions in problem size.

The CPU baseline typically computes connected components using breadth-first search (BFS). However, BFS-style traversal relies on dynamic frontiers and irregular memory access, which map poorly to GPU execution. Instead, we adopt a label propagation approach that exposes fine grained parallelism and avoids queue based traversal.

The algorithm 5 initializes each vertex with a unique label (its own index). In each iteration, a vertex updates its label to the minimum among its current label and those of its neighbors. This process continues until no label changes occur, at which point all vertices in the same connected component share the same label.

In Line 1, we initialize the `changed` flag to 1. This flag tracks whether any vertex's component status was updated during the previous kernel launch. The `componentStatus` array is initialized such that each vertex is its own component label. We then iteratively launch the `ComponentDecompose` kernel until no further updates occur and `changed` becomes zero (Lines 3–5).

In the kernel, each warp is assigned one vertex (Line 7). It reads the current component status (Line 8) and initializes a temporary variable to hold the minimum status (Line 9). The warp then iterates through the neighbors of the vertex, 32 at a time, and records the minimum status found among them (Lines 10–12). If a smaller status is found, the vertex's component status is updated and the `changed` flag is set to 1 via an atomic operation (Lines 14–16).

By avoiding BFS and relying on parallel label propagation, this method eliminates the need for explicit queues or frontier management. It achieves better memory coalescing and warp efficiency on GPUs. Moreover, its convergence behavior ensures correctness and fast termination in practice, especially for sparse graphs.

Algorithm 4 Clique Core Decomposition

Input: Graph $G = (V, E)$, clique size k, list of k-cliques
Output: Clique core values for all vertices

```
1. Allocate buffer and bufferTails arrays
2. Initialize count and level to 0
3. Initialize clique core values to initial clique degrees
4. while count < |V| do
5.     Launch SelectNodes kernel
6.     Launch ProcessNodes kernel
7.     level ← level +1
   ProcessNodes Kernel
8. if ThreadId == 0 then
9.     bufTail ← bufferTails[blockId]
10.    base ← 0
11. while True do
12.    loc ← base + warpId
13.    if loc ≥ bufTail then
14.        break
15.    if ThreadId == 0 then
16.        base ← base + TotalWarps
17.    v ← buffer[loc]
18.    for each lane in the warp do
19.        if vertex v is in the clique then
20.            for each vertex u in the clique do
21.                if v ≠ u then
22.                    a ← atomicSub(CliqueCore[u], 1)
23.                    if a == level +1 then
24.                        Add u to buffer
25.                        atomicAdd(count,1)
26.            CliqueStatus[laneId] ← level
```

6.5 Dynamic Exact Algorithm

We adopt the clique-induced flow formulation introduced in prior exact CPU based algorithms for k-clique DSD [7]. Our contribution is GPU oriented realization, compact flow network data structures, dynamic capacity updates across feasibility tests, and a parallel push-relabel solver that exploits the bipartite vertex-clique structure. These design choices enable repeated exact feasibility tests to scale efficiently on GPUs.

The network consists of three types of nodes: the original vertices V_C of a connected component, the set of t_C k-cliques (each represented as a node), and two auxiliary nodes: the source s_1 and the sink s_2. The connections are defined as follows:

- The source s_1 is connected to all vertices $v \in V_C$ with capacity $d(k, \psi)$.
- Each vertex $v \in V_C$ is connected to the sink s_2 with capacity $\alpha \cdot k$, where $\alpha = (u + l)/2$.
- Each clique node c is bidirectionally connected to the k vertices that form the corresponding k-clique with capacity $\infty, 1$ respectively.

This structure yields $|V_C| + t_C + 2$ nodes and $2|V_C| + 4kt_C$ directed edges, accounting for both forward and reverse residual connections, as shown in Figure 4 (c).

Algorithm 5 Connected Component Decomposition

Input: Graph $G = (V, E)$
Output: Component status of each vertex
1. Initialize changed $\leftarrow 1$
2. Initialize componentStatus[v] $\leftarrow v$ for all $v \in V$
3. **while** changed == 1 **do**
4. changed $\leftarrow 0$
5. Launch ComponentDecompose kernel

ComponentDecompose Kernel
6. **for each warp do**
7. $v \leftarrow$ warpId
8. $s \leftarrow$ componentStatus[v]
9. $s' \leftarrow s$
10. **for each neighbor** u **of** v **do**
11. $s_n \leftarrow$ componentStatus[u]
12. $s' \leftarrow \min(s', s_n)$
13. **if** laneId == 0 **then**
14. **if** $s' < s$ **then**
15. componentStatus[v] $\leftarrow s'$
16. atomicCAS(changed, 0, 1)

As shown in Algorithm 6, the proposed algorithm computes the maximum k-clique density through iterative max-flow evaluations on clique-induced networks. For each connected component, the algorithm initializes the lower and upper density bounds (l, u) (Lines 1-2), constructs a flow network using $\alpha = (u+l)/2$ (Line 4), floods all the reachable nodes from source (Line 6) and performs a GPU-based push–relabel procedure (Lines 8–10) to test the feasibility of α. If the maximum flow equals the total clique capacity $(t_C \times k)$ (Line 11-12), α is not feasible and the upper bound is reduced $(u \leftarrow \alpha)$; otherwise, the lower bound is increased $(l \leftarrow \alpha$, Line 14). This binary search continues until $(u - l) < 1/(|C|(|C| - 1))$ (Line 5), guaranteeing convergence to the exact density. The UpdateFlowNetwork kernel (Line 16) dynamically adjusts vertex–sink capacities between iterations, avoiding full reconstruction and maintaining high GPU throughput.

Initialization. Unlike classical push-relabel, which saturates all source edges and initializes all node heights to 0 (except the source), kernel *preFlow* perform a two-stage initialization. Flow is first pushed from the source s_1 to all vertices in V_C, then immediately drained to the sink s_2 where capacity allows. Afterwards, we set the height of all vertices in V_C who still have capacity left in edge connecting to sink to 1, while the source, sink, and clique nodes remain at height 0. This improves early excess propagation and reduces redundant relabel operations in the initial iterations.

Push-Relabel Variant. We adopt a parallel-friendly variant of the push-relabel algorithm. To accelerate convergence, we relax the classical admissibility condition: instead of requiring height[u] = height[v] + 1, we allow a push from u to any neighbor v satisfying height[u] > height[v]. This broader condition enables earlier discharge of excess and avoids excessive relabeling. Among admissible neighbors, we push to the one with minimum height, thereby directing flow along the steepest descent in label space, typically toward the sink. This selective pushing strategy avoids fragmentation of flow, reduces atomic contention, and improves memory efficiency.

Early Termination Criteria. To further accelerate convergence and avoid wasted computation, we introduce early stopping conditions based on structural properties of the flow network:

(1) **Zero Excess Condition:** Terminate if no vertex has remaining excess. This is the standard convergence condition for push-relabel and ensures correctness once all flow has reached the sink or is blocked.
(2) **Height-Based Pruning:** Terminate the processing of any vertex u with height$[u] > 2|V_C|$. Classical implementations use a loose upper bound of $|V_C| + t_C + 2$, equal to the total number of nodes. However, in our network, the longest alternating path from source to sink follows:

$$s_1 \rightarrow v_1 \leftrightarrow c_1 \leftrightarrow v_2 \leftrightarrow c_2 \leftrightarrow \cdots \leftrightarrow v_r \rightarrow s_2$$

This includes one edge from s_1, $2(r-1)$ intermediate hops between vertices and cliques, and one edge to s_2, for a total path length of $2r$. Since $r = |V_C|$, the worst-case height is $H_{\max} = 2|V_C|$. Any vertex exceeding this height is unreachable from the sink and can be pruned safely.
(3) **Sink Saturation Check:** Terminate if all edges from $v \in V_C$ to s_2 have zero residual capacity. Since only vertices are connected to the sink, flow can no longer reach the sink if these edges are fully saturated. This condition efficiently detects convergence in dense or highly saturated networks.

Combined, these criteria ensure correctness while eliminating unnecessary computation. In particular, the height-based pruning leverages the bipartite structure of the vertex-clique flow network to establish a tighter convergence threshold than traditional bounds, improving runtime without sacrificing accuracy.

Lightweight Global Relabeling. Classical push-relabel algorithms perform global relabeling every $|V_C| + t_C + 2$ steps by computing shortest-path distances from the sink via reverse BFS. In our case, we trigger relabeling every $2|V_C|$ steps, matching our derived height bound. Since reverse BFS is computationally expensive on GPUs due to irregular memory access and synchronization, we avoid it by using a structure-aware approximation.

We instead perform two localized push phases: first, from cliques to their connected vertices if residual capacity and excess exist, and second, from vertices to the sink under the same condition. After these pushes, we assign height 1 to all vertices that still have capacity to the sink; all other heights remain unchanged.

This selective relabeling encourages the flow to be pushed directly from source and cliques toward vertices that can still discharge excess to the sink. Assigning height 1 to such vertices ensures that they prioritize pushing flow to the sink, rather than back toward cliques or other non-terminal nodes, thereby accelerating convergence and reducing redundant work.

Although the global relabeling strategy uses a structure aware approximation, it affects only the scheduling of relabel operations, not the residual graph, capacities, or admissible pushes. All flow updates remain valid under the standard push-relabel framework, and the feasibility test is unchanged. Consequently, the algorithm remains exact and returns the same result as classical max-flow solvers, while converging faster. The performance impact of structure aware global relabeling is evaluated in Section 7.5 via an ablation study that disables this optimization and compares convergence behavior.

7 Experiments

In this section we will conduct experiments to evaluate the performance of our algorithm.

7.1 Setup

We evaluated our GPU-based implementation on NVIDIA A100 GPU (108 SMs, 40 GB memory). All CPU baselines are implemented in C++ to ensure a fair and consistent comparison. Specifically,

Algorithm 6 Dynamic Exact Algorithm

Input: Connected Components
Output: Maximum Clique Density $\rho(D*)$

1. Calculate the lower bound l, and set $\rho(D*)$ to l.
2. Calculate the upper bound u of all connected components
3. **for** `Each Component` C `in Connected Components` **do**
4. Create Flow Network F with $\alpha = \frac{u+l}{2}$
5. **while** $(u - l) > \frac{1}{|C|(|C|-1)}$ **do**
6. Launch `preFlow` Kernel to flood the vertices
7. `converged` $\leftarrow 0$
8. **while** ¬`converged` **do**
9. Launch `PushRelabel` Kernel
10. Launch `GlobalRelabl` kernel
11. **if** Max Flow is $t \times k$ **then**
12. $u \leftarrow \alpha$
13. **else**
14. $l \leftarrow \alpha$
15. $p(D*) \leftarrow l$
16. Launch `UpdateFlowNetwork` kernel to update α

Table 2. Graph Datasets

Graph	|V|	|E|	Max Degree
Yeast	1457	1945	1451
As 733	1486	3297	415
Net science	1588	2741	34
Autonomous Systems	6474	13895	1458
As Caida	6474	12572	1458
Ca Hepth	9877	51971	130
Grqc	26197	14490	81
Deezer	54573	498202	420
Co-authoring	108300	186936	558
Patent	6009555	16518947	793
Random Graph	500	6276	42
	2500	31256	45
	10000	124590	47
R-Mat	1000	94710	231

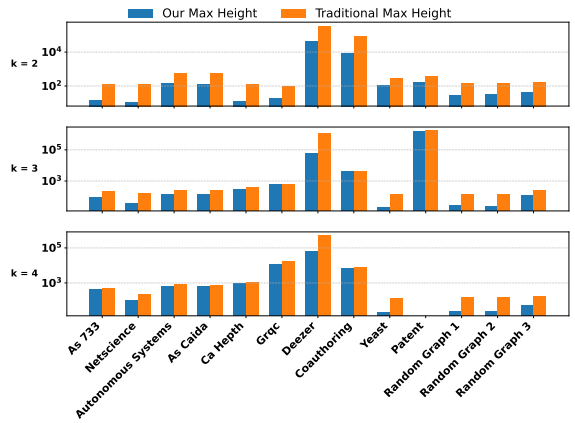


Fig. 5. Effect of Proposed Max Height

we first ported the original state-of-the-art exact CPU algorithm [7] from Java to C++. This serial C++ implementation serves as our primary CPU baseline. We further evaluate a variant of this CPU baseline augmented with our tightened density bounds to assess their impact in a serial CPU setting.

To our knowledge, no publicly available parallel CPU implementation exists for this problem. We implemented a parallel CPU version of the same high-level pipeline in C++, parallelizing component wise processing and clique enumeration where applicable, enabling a direct assessment of CPU side scalability. Several GPU-specific optimizations including revised height, and structure-aware relabeling, are tied to the push-relabel solver and cannot be meaningfully transferred to the Edmonds-Karp used in the CPU baselines.

All CPU experiments were conducted on Google Colab using an Intel Xeon CPU @ 2.20 GHz (8 virtual cores, 80 GB RAM). Our evaluation spans ten real-world graphs and four synthetic graphs

generated using Erdős–Rényi and R-MAT models (Table 2). Since no prior GPU-based exact method exists, this comparison provides a comprehensive view of CPU-GPU trade-offs.

7.2 Overall Performance

Table 3. Overall Performance

Graph	k	Density	CPU I	CPU II	PARALLEL	GPU
Yeast	2	2.26	202	152	118	115
	3	3.71	71	42	22	21
	4	2.71	56	51	23	22
	5	1.00	91	41	38	22
	6	0.17	91	40	29	20
As 733	2	8.19	33	21	24	15
	3	31.43	513	254	286	93
	4	68.67	5,146	4,405	4,467	415
	5	92.78	9,383	7,231	966	851
	6	79.37	10,474	9,328	7,621	913
Net science	2	9.50	11	11	14	11
	3	57.00	37	37	39	34
	4	242.25	100	98	100	98
	5	775.20	610	340	1,222	321
	6	1,938.00	924	544	20,194	143
AutonomousSystems	2	8.88	150	147	266	138
	3	35.91	533	501	450	143
	4	85.13	7,792	7,065	7,965	659
	5	126.77	23,729	22,755	1,912	1,600
	6	123.39	27,410	26,272	1,924	1,630
As Caida	2	8.56	100	104	166	135
	3	35.91	605	568	988	142
	4	85.13	8,120	7,529	7,138	646
	5	126.77	25,027	23,022	1,719	1,659
	6	123.39	30,075	26,117	1,969	1,721
Ca Hepth	2	31	186	180	557	13
	3	310	403	368	811	286
	4	2247.5	1,816	1,007	23,613	907
	5	12586	6,626	6,423	1,559,460	5,662
	6	37171	42,298	40,197	5,314,592	32,329
Grqc	2	22.39	1,275	695	4,615	20
	3	325.35	2,992	2,247	13,047	600
	4	3,450.24	686,432	1,513,800	3,218,810	720
	5	28,546.00	5,652,130	3,410,215	2,261,583	5,606
	6	191,120.00	777,817	542,403	431,285	360,970

Graph	k	Density	CPU I	CPU II	PARALLEL	GPU
Deezer	2	20.93	1,102	856	445	23
	3	84.79	351,346	211,666	471,509	23,243
	4	323.83	170,057	131,613	282,718	21,135
	5	714.45	372,969	156,258	695,729	19,601
	6	824.42	768,003	89,964	1,240,460	10,735
Co-authoring	2	28.84	27,543	26,406	53,578	18
	3	219.27	30,321	24,010	49,497	3,127
	4	1,261.07	40,588	20,523	63,217	6,817
	5	5,472.14	607,231	18,232	581,512	9,212
	6	19,683.80	28,792	17,194	16,251	15,285
Patent	2	50.85	9,876	8,432	7,690	128
	3	1111870	>5	>5	>5	828,280
	4	10,130.90	>5	>5	>5	655,368
	5	23,032.60	>5	>5	>5	408,506
	6	38,304.70	>5	>5	>5	100,701
Random Graph	2	12.55	2,047	1,996	113	28
	3	5.23	1,820	1,516	1,594	29
	4	1.00	62	41	46	24
	5	0.20	23	22	19	20
	6	0.00	19	19	19	18
	2	12.51	13,010	74,436	71,722	31
	3	1.21	1,419	1,655	381	23
	4	0.25	24	22	22	23
	5	0.00	37	36	34	21
	6	0.00	36	35	32	21
	2	12.46	548,383	2,394,180	292,904	41
	3	0.95	2,193	181	546	113
	4	0.25	135	129	289	54
	5	0.00	794	673	684	52
	6	0.00	853	614	691	53
RMAT	2	19.33	1,829	2,714	45	40
	3	83.73	45,571	46,097	45,517	914
	4	164.46	186,496	183,227	11,623	8,199
	5	265.92	249,533	256,628	237,916	83
	6	119.89	77,732	81,859	79,376	60

We configured the GPU kernel with 216 thread blocks, each containing 1024 threads. Table 3 presents a comparison between the execution times of the serial C++ implementation (CPU I), serial C++ implementation (CPU II) with new bounds, C++ parallel implementation (PARALLEL), and our GPU-based implementation across real-world datasets. Each dataset was evaluated for $k = 2$ to 6. A $> 5h$ represents exceeded the 5-hour time limit.

Overall, results highlight significant speedups achieved through GPU parallelism, demonstrating the scalability and efficiency of our approach. For small graphs such as *Yeast* and *As 733*, the GPU achieves 2–20× speedups, while for larger and denser networks (*Net Science*, *Autonomous Systems*, *Ca Hepth*), the acceleration grows to one or two orders of magnitude. The advantage becomes most evident on massive datasets like *Deezer*, *Co-authoring* and *Patent*, where the CPU version either exceeds the 5-hour limit or runs several orders of magnitude slower, while the GPU completes within seconds or minutes, yielding up to $10^4\times$ speedup.

7.3 Effect of Max Height

In Section 6.5, we proposed a new Height-Based pruning, with maximum height set $2 \cdot |V_C|$. It influences both the termination condition and the number of relabel cycles in each push-relabel

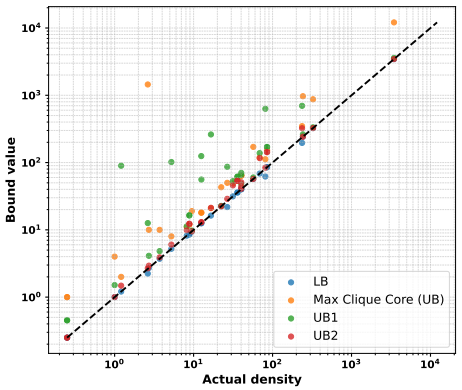


Fig. 6. Tightness of bounds relative to density.

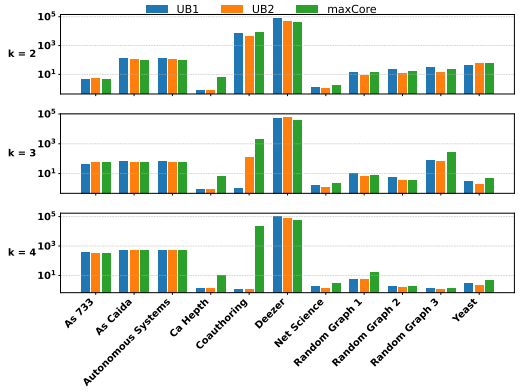


Fig. 7. Dynamic exact runtime under different bounds.

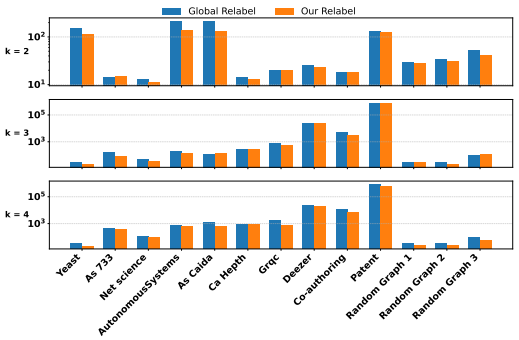


Fig. 8. Effect of structure aware global relabeling.

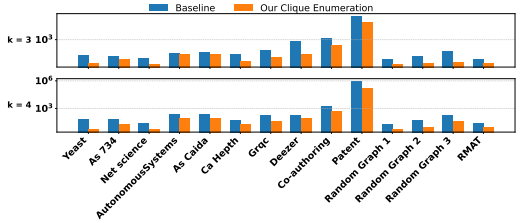


Fig. 9. Effect of clique enumeration design.

iteration. To evaluate its effect, we compared it against the traditional maximum height $|V_C| + t + 2$ in terms of execution time.

As shown in Figure 5, our proposed max height significantly reduces runtime across all datasets and k values. The gains are most pronounced on larger, denser graphs such as Deezer and grqc, where push-relabel typically develops deeper label hierarchies. Here, the tighter height bound avoids unnecessary pushes and speeds convergence. Overall, our max height performs consistently better (or comparably) to the traditional setting, indicating that a graph-aware max height improves efficiency while remaining robust for exact DSD computations.

7.4 Bound Comparison

We evaluate lower and upper bounds on the k-clique density across connected components from multiple graphs. Figure 6 compares each bound value (y-axis) against the actual density (x-axis) on a log–log scale; the dashed diagonal marks $y=x$, i.e., perfect tightness where the bound equals the actual value.

From the scatter plot, LB lies on or just below the $y = x$ line for most instances, indicating validity and frequent tightness. Among the upper bounds, UB_2 stays closest to the diagonal overall, making it the tightest across a wide density range. *MaxCore* and UB_1 exhibit greater dispersion, each is competitive in some cases but less consistently tight than UB_2. Overall, UB_2 is the most reliable upper bound, while LB provides a strong practical lower estimate.

Figure 7 shows how bounds affect the dynamic exact algorithm. Across all graphs and k, *UB2* achieves the lowest (or tied) runtime, sometimes by a large margin, reflecting the benefit of tighter bounds. *UB1* and *MaxCore* help in isolated scenarios but lack *UB2*'s consistency. In practice, we compute all three and use the minimum.

7.5 Effect of Structure Aware Global Relabeling

We conduct an ablation to isolate the impact of our structure aware global relabeling. We compare the full algorithm with a variant that uses a classical BFS-based relabel.

As shown in Figure 8, the structure aware relabel consistently reduces total runtime. It demonstrates that exploiting the flow network structure during relabeling is critical to achieving high performance in our solver.

7.6 Effect of Clique Enumeration Design

For the ablation study, we implement a lightweight variant (baseline) of our GPU clique enumeration that disables both canonical ordering and warp-local compaction. Instead of partitioning tasks and periodically flushing warp-local buffers into contiguous global arrays, this variant simply alternates between two global arrays: at each level, it reads partial cliques from one array and atomically appends the expanded partial cliques to the other. Consequently, no partitioning or Flush/compaction operation is used, and no canonical ordering constraints are enforced. We evaluate this variant against the full configuration and report the runtime of the clique enumeration stage only. We did not run the experiment for $k = 2$ since, in this case, the clique enumeration algorithm is bypassed entirely. Instead, we simply treat each edge of the directed acyclic graph as a 2-clique and copy them directly. Figure 9 shows that the full algorithm, using canonical ordering and warp local compaction, consistently achieves substantially lower enumeration time than the ablated variant, confirming that these GPU oriented design choices are critical to performance.

7.7 Load Imbalance Analysis

Our pipeline exhibits substantial load imbalance due to highly skewed structural properties in real-world graphs.

Clique enumeration task imbalance. During k-clique enumeration, each partial clique constitutes a task with associated metadata, a candidate set and, for each candidate, its valid neighbor set. Processing a task requires iterating over candidates and expanding them by scanning their valid neighbor sets, consequently, the cost of a task is proportional to the candidate size and the total size of valid neighbor sets. Figure 10 reports the distribution of task sizes (measured as the sum of candidate sizes and corresponding valid neighbor sizes), revealing substantial skew. We address this imbalance via warp-level task partitioning and a flush operation that compacts surviving tasks into contiguous arrays between iterations. This design enables dynamic, work-conserving execution, as soon as a warp finishes a task, it immediately acquires the next one, ensuring balanced utilization across warps.

Clique degree imbalance. Later stages, k-clique core decomposition and the max flow feasibility solver, are dominated by clique degree. In clique core decomposition, warps repeatedly select vertices whose clique degree matches the current peeling level and must process all incident cliques. Similarly, in the flow network solver, push operations traverse outgoing edges from active vertices, for vertex nodes, this includes edges to all incident clique nodes, making work proportional to clique degree. Figure 11 shows that clique degrees are also highly skewed, implying that static per-vertex assignment would again create stragglers. Accordingly, we employ the same dynamic

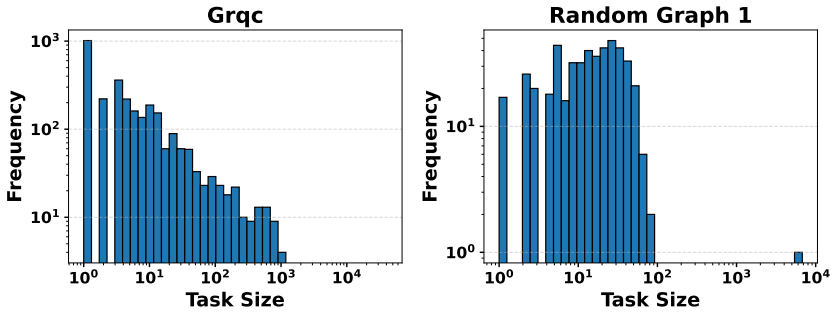


Fig. 10. Clique enumeration task size imbalance.

scheduling principle, warps fetch the next available active vertex (or excess vertex) using atomic work distribution and proceed asynchronously, maintaining high utilization under skew.

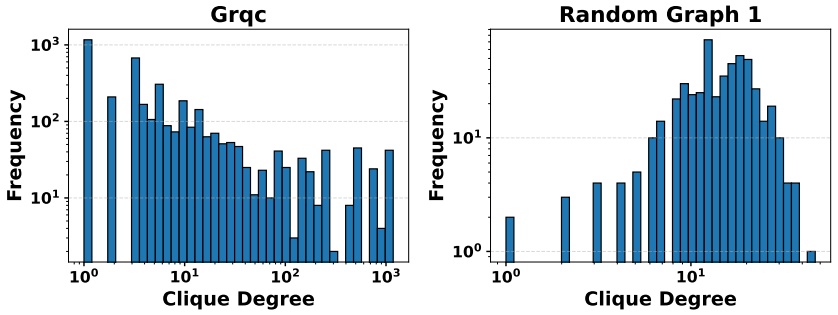


Fig. 11. Clique degree imbalance.

Overall, the experiments lead to the following conclusions. (1) our GPU implementation substantially outperforms both serial and parallel CPU baselines, with the largest gains appearing on larger graphs and higher k. (2) The proposed cap of $2|V_C|$ consistently reduces push-relabel runtime relative to the traditional setting, indicating that a graph-aware height limit accelerates convergence. (3) UB_2 is the tightest upper bound across a wide density range and yields the most consistent runtime improvements. (4) Structure-aware global relabeling provides consistent end-to-end reductions versus a classical BFS-based relabel variant. (5) Canonical ordering and warp-local compaction are key for fast enumeration, disabling them significantly increases enumeration time.

8 Conclusion

We introduced a fully GPU-accelerated framework for exact k-clique densest subgraph discovery, combining warp-level clique enumeration, clique-core and edge-based pruning, component-wise decomposition, and a structure-aware, vertex-centric push–relabel solver. Central to our design are compact GPU-friendly data structures, a maximum-height cap tailored to clique-induced networks, and a tighter, component-specific upper bound on clique density that shrinks the binary-search interval and reduces the number of flow evaluations.

Across diverse real and synthetic graphs, our system delivers substantial end-to-end speedups over the serial and parallel CPU baseline while producing identical solutions. Gains are most pronounced on larger and denser graphs and for higher k, where our algorithm and tighter bounds maximize parallelism and cut redundant computation.

ACKNOWLEDGMENTS

This work was supported by NSF CRII-2245792.

References

[1] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. 2016. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications* 60 (2016), 19–31.

[2] Anna Arpaci-Dusseau, Zixiang Zhou, and Xuhao Chen. 2024. Accurate and Fast Approximate Graph Pattern Mining at Scale. *Proc. VLDB Endow.* 18, 2 (2024), 93–107.

[3] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577. doi:10.1145/362342.362367

[4] Jingyu Chen and Yousef Saad. 2012. Dense Subgraph Extraction with Application to Community Detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24, 7 (2012), 1216–1230. doi:10.1109/TKDE.2011.66

[5] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355. doi:10.1137/S009753970240309X

[6] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (1972), 248–264.

[7] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *PVLDB (Proceedings of the VLDB Endowment)* 12, 11 (2019), 1719–1732. doi:10.14778/3342263.3342645

[8] Eric Fratkin, Branden T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. 2006. MotifCut: Regulatory Motifs Finding with Maximum Density Subgraphs. *Bioinformatics* 22, 14 (2006), e150–e157. doi:10.1093/bioinformatics/btl244

[9] Andrew V. Goldberg. 1984. *Finding a Maximum Density Subgraph.* Technical Report UCB/CSD-84-171. University of California, Berkeley.

[10] Andrew V. Goldberg and Robert E. Tarjan. 1988. A New Approach to the Maximum-Flow Problem. *J. ACM* 35, 4 (1988), 921–940.

[11] Godfrey H Hardy, John E Littlewood, and George Pólya. 1952. *Inequalities* (2 ed.). Cambridge University Press.

[12] Kuan-Hsun Hsieh et al. 2024. Engineering a Workload-Balanced Push-Relabel Algorithm for Massive Graphs on GPUs. *arXiv preprint arXiv:2404.00270* (2024).

[13] Samir Khuller and Barna Saha. 2009. On Finding Dense Subgraphs. In *Automata, Languages and Programming (ICALP 2009) (Lecture Notes in Computer Science, Vol. 5555)*. Springer, 597–608. doi:10.1007/978-3-642-02930-1_49

[14] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzone, and Francesco Bonchi. 2024. A survey on the densest subgraph problem and its variants. *Comput. Surveys* 56, 8 (2024), 1–40.

[15] Alfréd Rényi. 1961. On measures of entropy and information. *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics* (1961), 547–561.

[16] Barna Saha, Stephen Hoch, Samir Khuller, Louiqa Raschid, and Xuan-Nam Zhang. 2010. Dense Subgraphs with Restrictions and Applications to Gene Annotation Graphs. In *Proceedings of the 14th International Conference on Research in Computational Molecular Biology (RECOMB) (Lecture Notes in Computer Science, Vol. 6044)*. Springer, 456–472. doi:10.1007/978-3-642-12683-3_31

[17] Roman Samusevich, Matthieu Danisch, and Mauro Sozio. 2016. Local Triangle-Densest Subgraphs. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 33–40. doi:10.1109/ASONAM.2016.7752211

[18] Jathin Soman, Kishore Kothapalli, and Dinesh Patidar. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*. IEEE, 1–8. doi:10.1109/IPDPSW.2010.5470882

[19] Charalampos E. Tsourakakis. 2015. The k-Clique Densest Subgraph Problem. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*. International World Wide Web Conferences Steering Committee, 1122–1132. doi:10.1145/2736277.2741130

[20] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser Than the Densest Subgraph: Extracting Optimal Quasi-Cliques with Quality Guarantees. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 104–112. doi:10.1145/2487575.2487581

[21] Ying Zhang and Srinivasan Parthasarathy. 2012. Extracting, Analyzing and Visualizing Triangle k-Core Motifs within Networks. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE)*. IEEE, 1049–1060. doi:10.1109/ICDE.2012.110

[22] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *Proceedings of the VLDB Endowment (PVLDB)* 6, 2 (2012), 85–96. doi:10.14778/2535568.2448930

Received October 2025; revised January 2026; accepted February 2026