GPU-Accelerated η-threshold Decomposition for Uncertain Graphs

Yu Chen
Zhejiang University
cy1098@zju.edu.cn

Chong Liu
Zhejiang University
lc_jxyy@zju.edu.cn

Qing Liu
Zhejiang University
qingliucs@zju.edu.cn

Zhonggen Li
Zhejiang University
zgli@zju.edu.cn

Yifan Zhu
Zhejiang University
xtf_z@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

ABSTRACT

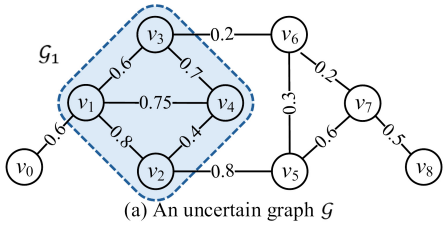
The η-threshold decomposition in uncertain graphs, which calculates the η-thresholds for each vertex, is a fundamental problem in graph analysis. However, the current CPU-based peeling algorithm suffers from prohibitive computational costs, making it infeasible for time-sensitive applications such as fraud detection and dynamic public opinion monitoring. To address this, we introduce Gatd, the first GPU-accelerated framework for η-threshold decomposition, co-designed with GPU architecture to enable efficient parallelization. Given that the problem is a computationally intensive per-vertex task dominated by probability computation, thereby constraining efficiency, Gatd incorporates three enhancement modules: (i) Redundancy reduction through lower-bound pruning, leveraging safety thresholds from prior iterations, and batch updating of vertices sharing the same η-threshold; (ii) Adaptive parallelization utilizing dynamically sized thread collaboration groups and hybrid scheduling to match computational resources with dynamic workloads; and (iii) Three-stage load balancing based on neighbor grouping, work stealing, and hierarchical merging to mitigate supernode-induced imbalance. Extensive experiments on diverse uncertain graphs demonstrate that the optimized Gatd achieves speedups of up to four orders of magnitude over state-of-the-art CPU-based methods and existing GPU-based graph processing frameworks, facilitating efficient decomposition even for large-scale networks.

PVLDB Reference Format:
Yu Chen, Chong Liu, Qing Liu, Zhonggen Li, Yifan Zhu, and Yunjun Gao. GPU-Accelerated η-threshold Decomposition for Uncertain Graphs. PVLDB, 19(10): 2591-2603, 2026.
doi:10.14778/3828612.3828617

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AbigailJixiangyuyu/uco-gpu.

1 INTRODUCTION

Many real-world networks, including Protein-Protein Interaction (PPI) networks [11, 22], financial transaction networks [5, 17], and social networks [45], exhibit inherent uncertainty, where edges are associated with probabilities of existence, resulting in uncertain graphs. Mining cohesive subgraphs in graph analysis is a fundamental problem that has received much attention in data mining and engineering fields [13, 15, 21, 39]. Numerous cohesive subgraph models have been proposed for uncertain graphs, such as core-based models [2, 25, 50], truss-based models [14, 52], and clique-based models [20, 26, 27].



(a) An uncertain graph $\mathcal{G}$

k	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	0.6	0.88	0.88	0.88	0.88	0.8	0.552	0.6	0.5
2	-	0.42	0.32	0.42	0.42	0.136	0.136	0.136	-

(b) η-threshold decomposition result of $\mathcal{G}$

Figure 1: A Demonstration Example

This paper primarily focuses on the (k, η)-core model [2]. Given an integer k and a probabilistic threshold η, a (k, η)-core is a maximal subgraph where each vertex's probability of having a degree of at least k is not less than η. For example, in the uncertain graph $\mathcal{G}$ shown in Figure 1(a), the subgraph induced by vertices $\{v_1, v_2, v_3, v_4\}$ forms a $(2, 0.32)$-core. Given an uncertain graph $\mathcal{G}$ and an integer k, the η-threshold of a vertex $u \in \mathcal{G}$ w.r.t. k is the maximum η value such that u is contained in a (k, η)-core. Note that the η of η-threshold is just a symbol, not a specific number. For instance, in Figure 1(a), v_6 belongs to a $(1, 0.552)$-core, but not to any $(1, \eta)$-core with $\eta > 0.552$; thus, its η-threshold w.r.t. $k = 1$ is 0.552. Correspondingly, the η-threshold decomposition of $\mathcal{G}$ aims to calculate the η-threshold for each vertex w.r.t. all possible k values. Figure 1(b) illustrates the η-threshold decomposition result of $\mathcal{G}$.

The η-threshold decomposition of uncertain graphs has a wide range of applications. For example, in PPI networks, vertices with high η-thresholds help identify functional protein complexes [11, 32]. In financial transaction networks, such vertices often correspond to accounts involved in suspicious transaction patterns, aiding the detection of fraud rings [48] and money laundering organizations [4]. In social networks, users with elevated η-thresholds

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828617

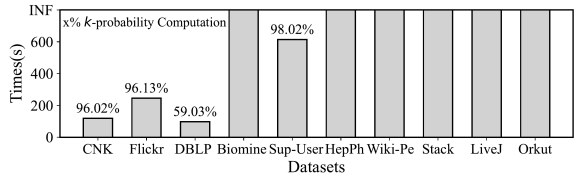


Figure 2: Runtime of the existing CPU-based Method

typically play central roles in influential communities, making them valuable for applications such as personalized recommendation [42], targeted advertising [23], and public opinion control [12].

Motivations. (**1**) Current η-threshold decomposition methods rely exclusively on a CPU-based peeling framework [41, 44], which suffers from severe computational inefficiency. Specifically, for each possible k, the CPU-based method iteratively removes the vertex u with the minimum probability that u's degree is at least k, then updates the probabilities of u's neighbors, until no further vertices can be removed. The overall time complexity is $O(k_{max} \cdot n \cdot \log n + k_{max}^2 \cdot m \cdot d_{max})$, where m, n, k_{max}, and d_{max} denote the number of edges, the number of vertices, the maximum k value, and the maximum degree of the uncertain graph, respectively. As shown in Figure 2, the CPU-based method exceeds 12 hours on most benchmark datasets, rendering it impractical for *time-sensitive applications* like fraud detection [48] in financial transaction networks and dynamic public opinion monitoring [12] on social platforms. Its excessive runtime also hinders offline exploratory analysis, such as protein complex identification [11, 32] in PPI networks. (**2**) Although Graphics Processing Units (GPUs) have excelled in accelerating various graph tasks [1, 35, 36], existing GPU-based graph processing methods or systems are ill-suited for η-threshold decomposition due to intensive probability computations. These computations can exacerbate thread divergence and load imbalance within GPUs' inherent single instruction, multiple threads (SIMT) execution model. These two limitations emphasize critical necessity for more efficient GPU-based algorithms tailored for η-threshold decomposition on large-scale uncertain graphs.

Challenges. As previously highlighted, *η-threshold decomposition involves intensive per-vertex computation*: each vertex iteratively computes its *k-probability*, defined as the probability that its degree is at least k, via dynamic programming with a time complexity of $O(k \cdot d)$, where d is the vertex's degree. For example, Figure 2 indicates the ratio of k-probability computation to total decomposition time on partial datasets, peaking at 98.02% on Sup-User. *This demanding per-vertex computation poses a unique and significant challenge for efficient GPU parallelization.* Moreover, accelerating η-threshold decomposition on GPUs faces two major challenges. *First, the workload is highly dynamic.* The current method processes all $k \in [1, k_{max}]$, where the computational cost of k-probability increases with the parameter k, leading to varying per-vertex workloads. For each k, it iteratively deletes the vertex with minimum k-probability (min-k-probability) and updates the k-probabilities of its neighbors. The number of affected neighbors varies significantly across iterations, hindering sustained high parallelism. *Second, severe load imbalance arises from power-law degree distributions* [18]. Real-world graphs typically contain a few high-degree *supernodes* [38], whose k-probability computation is orders of magnitude more expensive than that of low-degree vertices, making them become bottlenecks. This leads to notable thread divergence [35], limiting throughput and scalability.

Our Methods. To address these challenges, we propose **Gatd**, **the first GPU-accelerated framework** tailored for η-threshold decomposition. Rather than trivially adapting CPU-based methods, Gatd fundamentally restructures the execution model to mitigate the bottlenecks associated with intensive per-vertex computations and the sequential min-k-probability search. Specifically, it employs two specialized kernels: a *Parallel k-Probability Computation* kernel that distributes collaborative dynamic programming across intra-warp threads, and a *Min-k-probability Reduction* kernel that identifies the global minimum vertex via hierarchical parallel reduction. Building upon this architecture, our core technical contributions are summarized as follows:

- **Redundancy Reduction.** To reduce unnecessary probability computations, we introduce lower-bound pruning and batch updating techniques. By leveraging the theoretical monotonicity of η-thresholds, we derive a computation-free, safe lower bound from prior iterations. Coupled with batch updating, this strategy guarantees that each affected neighbor's k-probability is updated only once per iteration.
- **Adaptive Parallelism.** To maximize hardware utilization under highly dynamic workloads, we introduce the fine-grained parallelization scheme leveraging Thread Cooperative Groups (TCGs). Complemented by a degree-aware hybrid scheduling strategy, this approach dynamically matches GPU execution units (e.g., specific TCG sizes or full blocks) to the fluctuating k and workload scales, effectively mitigating thread divergence.
- **Three-stage Load Balancing.** To resolve the severe straggler effect caused by supernodes, we devise a novel three-stage load balancing mechanism. By integrating neighbor grouping, global device-side work stealing, and hierarchical two-way merging, this mechanism decomposes expensive sequential per-vertex computations into balanced, fully parallelizable subtasks across streaming multiprocessors.
- **Extensive Experiments.** We conduct comprehensive evaluations on diverse real and semi-real uncertain graphs. The results demonstrate that Gatd achieves up to four orders of magnitude speedup over state-of-the-art CPU baselines, validating the efficacy and scalability of our framework.

Roadmap. Section 2 formalizes the problem and outlines the CPU-based algorithm along with GPU architecture. Sections 3 and 4 present our GPU-accelerated framework and three optimization techniques. Experimental results are reported in Section 5. Finally, Section 6 reviews related work and Section 7 concludes the paper.

2 PRELIMINARIES

This section introduces the η-threshold decomposition problem, reviews an established CPU-based solution, and presents an overview of GPU architecture.

2.1 Problem Definition

We begin with introducing k-core and core number in deterministic graphs. Let $G = (V_G, E_G)$ be an undirected deterministic graph, where V_G and E_G denote the sets of vertices and edges, respectively.

Given a vertex $u \in V_G$, the degree of u is defined as $deg(u, G) = |\{v \in V_G | (u, v) \in E_G\}|$.

Definition 2.1 (k-**core**, **Core Number**). *Given a graph G and an integer k, the k-core of G is a maximal subgraph $H = (V_H, E_H) \subseteq G$ in which each vertex has a degree of at least k, i.e., $\forall u \in V_H$, $deg(u, H) \geq k$. The core number of a vertex u, denoted by $c(u)$, is the maximum k such that there exists a non-empty k-core containing u.*

For example, in Figure 1(a), if we ignore edge probabilities and take the graph as a deterministic graph, all vertices except v_0 and v_8 constitute the 2-core. Since v_0 and v_8 only belong to 1-core, their core numbers are both 1.

Building on the aforementioned concepts, we formalize the η-threshold decomposition problem for uncertain graphs. Let $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, p)$ be an undirected uncertain graph, where p is a function assigning an existence probability $p_e \in (0, 1]$ to each edge $e \in E_\mathcal{G}$. We adopt the standard independent uncertain graph model [2], assuming mutually independent edge probabilities, for two main reasons. First, our objective is to accelerate η-threshold decomposition on uncertain graphs using GPUs. All existing studies on η-threshold decomposition employ the independent uncertain graph model, which provides a consistent and well-established baseline for performance comparison. Second, we acknowledge that real-world graphs (e.g., transaction networks) often involve correlated edges. However, η-threshold decomposition over correlated uncertain graphs [46] requires fundamentally new algorithm designs rather than simple extensions, which goes beyond the scope of this work and is deferred to our future studies.

Under this independent assumption, we adopt the well-known possible-world semantics [9] for uncertain graph analysis. Specifically, the deterministic graph $G = (V_G, E_G) \sqsubseteq \mathcal{G}$ represents a possible world of $\mathcal{G}$, where $V_G = V_\mathcal{G}$ and $E_G \subseteq E_\mathcal{G}$. The probability of a graph instance G is denoted as:

$$Pr(G) = \prod_{e \in E_G} p_e \cdot \prod_{e \in E_\mathcal{G} \setminus E_G} (1 - p_e) \tag{1}$$

Totally, an uncertain graph $\mathcal{G}$ has $2^{|E_\mathcal{G}|}$ possible worlds. Let $\mathcal{G}^u_{\geq k}$ be the set of all possible worlds of $\mathcal{G}$ containing u, where u's degree is no less than k, i.e., $\mathcal{G}^u_{\geq k} = \{G \sqsubseteq \mathcal{G} | deg(u, G) \geq k\}$. Based on it, we define the k-probability as follows.

Definition 2.2 (k-**probability** [44]). *Given an uncertain graph $\mathcal{G}$ and an integer k, the k-probability of a vertex $u \in V_\mathcal{G}$, denoted by $k\text{-}prob(u, \mathcal{G})$, is the probability that u's degree is not less than k, i.e., $k\text{-}prob(u, \mathcal{G}) = Pr[deg(u, \mathcal{G}) \geq k] = \sum_{G \in \mathcal{G}^u_{\geq k}} Pr(G)$.*

Consider vertex v_6 in Figure 1(a). $1\text{-}prob(v_6, \mathcal{G}) = 1 - 0.8 \times 0.7 \times 0.8 = 0.552$ and $3\text{-}prob(v_6, \mathcal{G}) = 0.2 \times 0.3 \times 0.2 = 0.012$. We use $k\text{-}prob(u)$ instead of $k\text{-}prob(u, \mathcal{G})$ when the context is clear. Next, we introduce the (k, η)-core for uncertain graphs.

Definition 2.3 ((k, η)-**core** [2]). *Given an uncertain graph $\mathcal{G}$, an integer k, and a probabilistic threshold $\eta \in [0, 1]$, the (k, η)-core is a maximal subgraph $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, p) \sqsubseteq \mathcal{G}$ satisfying: for each vertex $u \in V_\mathcal{H}$, the probability that u has a degree of at least k in $\mathcal{H}$ is no less than η, i.e., $\forall u \in V_\mathcal{H}$, $k\text{-}prob(u, \mathcal{H}) \geq \eta$.*

For instance, in Figure 1(a), with $k = 2$ and $\eta = 0.3$, the subgraph $\mathcal{G}_1$ forms a (2, 0.3)-core since (i) 2-prob(v_1, $\mathcal{G}_1$) = 0.81, 2-prob(v_2, $\mathcal{G}_1$) = 0.32, 2-prob(v_3, $\mathcal{G}_1$) = 0.42, and 2-prob(v_4, $\mathcal{G}_1$) = 0.658, and (ii) $\forall \mathcal{G}' \subseteq \mathcal{G}$, $\forall v \in \mathcal{G} - \mathcal{G}_1$, 2-prob($v$, $\mathcal{G}'$) < 0.3. We would like to highlight three points. (1) When $\eta = 0$, the (k, η)-core is equivalent to the standard k-core of the underlying deterministic graph, disregarding edge probabilities of $\mathcal{G}$. Hence, we assume $\eta > 0$ in the remainder of the paper. (2) For a vertex $u \in V_\mathcal{G}$, if $k > c(u)$, then u cannot belong to any (k, η)-core with $\eta > 0$. (3) Like k-core, the (k, η)-core also has the nesting property: for an integer k and two probabilistic thresholds $0 < \eta_1 \leq \eta_2 \leq 1$, the (k, η_1)-core contains (k, η_2)-core. Based on this property, η-threshold is defined as follows.

Definition 2.4 (η-**threshold** [44]). *Given an uncertain graph $\mathcal{G}$, an integer k, and a vertex $u \in V_\mathcal{G}$, the η-threshold of u w.r.t. k, denoted by $\eta(k, u)$, is the maximum η such that there is a non-empty (k, η)-core containing u.*

Equivalently, the (k, η)-core can be characterized as the maximal induced subgraph containing all vertices u such that $\eta(k, u) \geq \eta$. For example, in Figure 1(a), when $k = 2$ and $\eta = 0.3$, since $\eta(2, v_1) = \eta(2, v_3) = \eta(2, v_4) = 0.42 > 0.3$ and $\eta(2, v_2) = 0.32 > 0.3$, the $(2, 0.3)$-core is the subgraph $\mathcal{G}_1$. We next provide a formal definition of the η-threshold decomposition problem in uncertain graphs.

Problem 1 (η-**threshold Decomposition** [44]). *Given an uncertain graph $\mathcal{G}$, the η-threshold decomposition of $\mathcal{G}$ is to calculate $\eta(k, u)$ for $\forall u \in V_\mathcal{G}$ w.r.t. $\forall k \in [1, c(u)]$.*

For instance, Figure 1(b) shows the η-threshold decomposition result of $\mathcal{G}$. Take the vertex v_6 as an example, since $c(v_6) = 2$, we only need to compute $\eta(1, v_6) = 0.552$ and $\eta(2, v_6) = 0.136$.

2.2 CPU-Based Decomposition Algorithm

For Problem 1, we first introduce the state-of-the-art CPU-based η-threshold decomposition (CTD) algorithm [44], which serves as the foundation for our GPU-accelerated approach. CTD processes each possible k sequentially via a peeling framework. For a given k, it iteratively deletes the vertex u with the minimum k-probability, sets $\eta(k, u)$ accordingly, and updates the k-probabilities of its neighbors until no vertices remain. In this process, the k-probability of a vertex u in an uncertain graph $\mathcal{G}$ is defined as:

$$k\text{-}prob(u, \mathcal{G}) = \sum_{i=k}^{deg(u,\mathcal{G})} Pr[deg(u, \mathcal{G}) = i] = 1 - \sum_{i=0}^{k-1} Pr[deg(u, \mathcal{G}) = i] \tag{2}$$

Here, $deg(u, \mathcal{G})$ represents the degree of u in $\mathcal{G}$. Following Equation 2, we initialize $0\text{-}prob(u, \mathcal{G}) = 1$, and iteratively compute: $i\text{-}prob(u, \mathcal{G}) = (i-1)\text{-}prob(u, \mathcal{G}) - Pr[deg(u, \mathcal{G}) = (i-1)]$ for $i = 1$ to k. The CTD algorithm then employs the dynamic programming approach [2] to calculate $Pr[deg(u, \mathcal{G}) = i]$ for all $i \in [0, k-1]$. For brevity, we abbreviate $deg(u, \mathcal{G})$ to $d(u)$. Let $E(u) = \{e_1, e_2, \ldots, e_{d(u)}\}$ be the edges incident to u in a specific order. The key observation is that, if u has a degree of i, one of two cases is true: (1) u has $i-1$ edges in $\{e_1, \ldots, e_{d(u)-1}\}$ and edge $e_{d(u)}$ exists; or (2) u has i edges in $\{e_1, \ldots, e_{d(u)-1}\}$ and edge $e_{d(u)}$ does not exist.

Given a subset $E'(u) = \{e_1, \ldots, e_h\} \subseteq E(u)$ of edges incident to vertex u, define the subgraph $\mathcal{G}'(u) = (V_\mathcal{G}, E_\mathcal{G} \setminus \{e_{h+1}, \ldots, e_{d(u)}\}, p)$. Let the *state array* $X(h, j)$ denote the probability that u has degree j in $\mathcal{G}'(u)$, i.e., $Pr[deg(u, \mathcal{G}'(u)) = j]$. Based on this formulation,

Algorithm 1: CPU-based η-threshold Decomposition Algorithm (CTD) adapted from [44]

Input: An uncertain graph $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}}, p)$
Output: All $\eta(k,u)$ for $\forall u \in V_{\mathcal{G}}$ w.r.t. $k \in [1, c(u)]$

1 compute $c(u)$ for each $u \in V_{\mathcal{G}}$ using Algorithm in [44];
2 $k_{max} \leftarrow \max_{u \in V_{\mathcal{G}}} c(u)$;
3 **for** $k \in [k_{max}, 1]$ **do**
4 $\mathcal{G}' = (V_{\mathcal{G}'}, E_{\mathcal{G}'}, p) \leftarrow \mathcal{G}[\{u \in V_{\mathcal{G}} | c(u) \geq k\}]$;
5 **for** $\forall u \in V_{\mathcal{G}'}$ **do**
6 initialize $k\text{-}prob(u, \mathcal{G}')$ incrementally;
7 $curT \leftarrow 0$;
8 **while** $\mathcal{G}'$ *is not empty* **do**
9 $u \leftarrow \arg\min_{v \in V_{\mathcal{G}'}} k\text{-}prob(v, \mathcal{G}')$;
10 $curT \leftarrow \max\{k\text{-}prob(u, \mathcal{G}'), curT\}$;
11 $\eta(k,u) = curT$;
12 $V_{\mathcal{G}'} \leftarrow V_{\mathcal{G}'} \setminus \{u\}$;
13 **for** $\forall (u,v) \in E_{\mathcal{G}'}$ **do**
14 update $k\text{-}prob(v, \mathcal{G}')$ according to Equation 2;
15 **return** *all η-thresholds.*

for all $h \in [1, d(u)]$ and $j \in [0, i]$(where $i \leq h$), the following recurrence equation holds:

$$X(h,j) = p_{e_h} \cdot X(h-1, j-1) + (1 - p_{e_h}) \cdot X(h-1, j) \quad (3)$$

The initialization is defined as $X(0,0) = 1$, $X(h,-1) = 0$ for $\forall h \in [0, d(u)]$, and $X(0,j) = 0$ for $\forall j \in [1, i]$. By combining equations 2 and 3, we have $Pr[deg(u, \mathcal{G}) = (k-1)] = X(d(u), k-1)$. Once calculated, this yields all $Pr[deg(u, \mathcal{G}) = i]$ for $i \in [0, k-1]$, enabling the derivation of $k\text{-}prob(u, \mathcal{G})$. The time complexity for computing u's k-probability is $O(k \cdot d(u))$.

Based on the above introduction, the CTD algorithm is summarized in Algorithm 1. It initially employs the classic core decomposition algorithm to compute the core number $c(u)$ of each vertex u (line 1). Then, for k decreasing from k_{max} to 1, CTD refines the uncertain graph to $\mathcal{G}'$, which is the induced subgraph of $V_{\mathcal{G}'} = \{u \in V_{\mathcal{G}} | c(u) \geq k\}$ in $\mathcal{G}$, as only these vertices can belong to a (k,η)-core with $\eta > 0$. The algorithm (lines 2-6) initializes the k-probability of $u \in V_{\mathcal{G}'}$ incrementally by reusing results from the $(k+1)$-th iteration and considering only for newly included neighbors v (where $c(v) = k$). After initialization, the vertex u with the minimum k-probability is removed, its η-threshold $\eta(k,u)$ is recorded, and the k-probabilities of its neighbors are updated via Equation 3 (lines 9-14). This process continues until all vertices in $\mathcal{G}'$ are processed. The overall time complexity is $O(k_{max} \cdot n \cdot \log n + k_{max}^2 \cdot m \cdot d_{max})$, where $n = |V_{\mathcal{G}}|$, $m = |E_{\mathcal{G}}|$, and d_{max} denotes the maximum degree of $\mathcal{G}$. This approach is computationally expensive, highlighting the necessity for more efficient η-threshold decomposition algorithms.

2.3 GPU Architecture

Graphics processing units (GPUs) feature a massively parallel architecture capable of concurrently executing thousands of threads under high memory bandwidth. A GPU comprises multiple streaming multiprocessors (SMs), each operating in single-instruction, multiple-threads (SIMT) mode. The minimum execution unit within an SM is a warp, consisting of 32 threads that execute the same instruction in lockstep. When threads within a warp diverge into different branches (e.g., if-else statements), the hardware serializes branch execution, using masking to disable/enable threads as needed. This phenomenon, termed warp divergence [37], is a well-documented bottleneck in GPU computing.

Table 1: Summary of GPU-Related Notations

Notation	Definition
$blockNum$	Number of thread blocks in the kernel grid
$warpNum$	Number of warps per thread block
$warpSize$	Number of threads per warp
$threadNum$	Total number of threads ($threadNum = blockNum \cdot warpNum \cdot warpSize$)
$threadId$	Local thread ID within a block ($threadId \in [0, warpNum \cdot warpSize)$)
$laneId$	Lane index within a warp ($laneId = threadId \bmod warpSize$)
$warpId$	Local warp ID within a block ($warpId = \lfloor threadId / warpSize \rfloor \in [0, warpNum)$)

In the NVIDIA CUDA programming model, GPU computation is expressed through kernels: developer-defined functions that specify parallel execution patterns. A kernel is launched from a host (CPU) program, which explicitly defines its execution configuration as a grid of thread blocks. Each block comprises a fixed number of threads that execute the same kernel on distinct data elements. At runtime, blocks are dynamically scheduled onto available SMs, where threads are grouped into warps and executed in SIMT mode. Multiple blocks can reside concurrently on a single SM, with new blocks launched as resources become available. However, when threads or blocks process workloads of varying sizes, such as vertices with highly skewed degrees [28], load imbalance may arise across warps and SMs, fundamentally throttling parallel efficiency.

For clarity and consistency, Table 1 summarizes the GPU-related notations, including structural dimensions (e.g., numbers of blocks and warps) and thread-level identifiers for workload distribution.

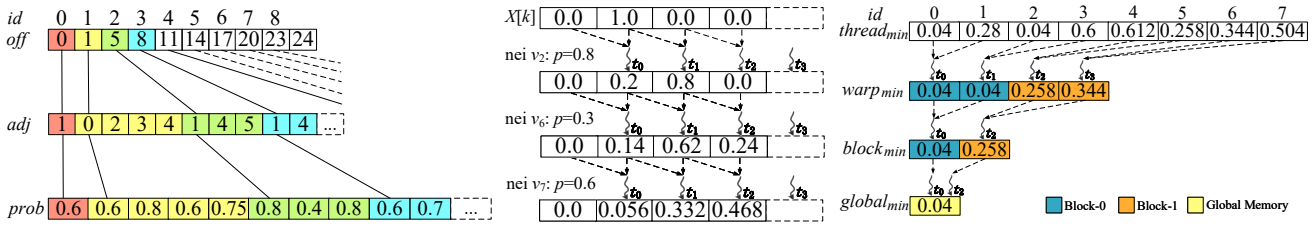
3 GPU-ACCELERATED DECOMPOSITION ALGORITHM

This section details Gatd, our GPU-accelerated η-threshold decomposition algorithm. Departing from CPU paradigms, Gatd restructures the execution model via two specialized kernels: **(i)** *Parallel k-Probability Computation* (Section 3.2), mitigating per-vertex overhead through intra-warp collaborative dynamic programming; and **(ii)** *Min-k-probability Reduction* (Section 3.3), eliminating the sequential search bottleneck via hierarchical parallel reduction.

3.1 Algorithm Overview

Uncertain Graph Representation. GPU memory capacity often limits the scale of graphs processable by GPU-based algorithms. Thus, a compact graph representation is essential. To this end, we adopt the widely used Compressed Sparse Row (CSR) [3] format to store the uncertain graph $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}}, p)$ with three arrays:

- ***off***: An offset array of length $|V_{\mathcal{G}}| + 1$, where the neighbors of vertex $u \in V_{\mathcal{G}}$ are sorted in ***adj*** from index $p_s = off[u]$ to $p_e = off[u+1] - 1$.



(a) CSR-based uncertain graph representation (b) Parallel k-probability computation for v_5 (c) The reduction of min-k-probability vertex

Figure 3: A Walk-through Example of Gatd

Algorithm 2: GPU-accelerated η-threshold Decomposition Algorithm (Gatd)

Input: An uncertain graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, p)$
Output: All $\eta(k,u)$ for $u \in V_\mathcal{G}$ w.r.t. $k \in [1, c(u)]$

1 parallel computing $c(u)$ for $u \in V_\mathcal{G}$ using Algorithm in [1];
2 $k_{max} \leftarrow \max_{u \in V_\mathcal{G}} c(u)$;
3 **for** $k \in [k_{max}, 1]$ **do**
4 $\quad \mathcal{G}' = (V_{\mathcal{G}'}, E_{\mathcal{G}'}, p) \leftarrow \mathcal{G}[\{u \in V_\mathcal{G} | c(u) \geq k\}]$;
5 $\quad$ *initialize* k-$prob(u, \mathcal{G}') \leftarrow 1.0$ ***for*** $\forall u \in V_{\mathcal{G}'}$;
6 $\quad$ *launch kernel: ComputeKProb*$(V_{\mathcal{G}'}, k)$; // *Algorithm 3*
7 $\quad curT \leftarrow 0$;
8 $\quad$ **while** $\mathcal{G}'$ *is not empty* **do**
9 $\quad\quad u \leftarrow$ *launch kernel: ReduceMinKProb*$(V_{\mathcal{G}'})$;
10 $\quad\quad curT \leftarrow \max\{k\text{-}prob(u, \mathcal{G}'), curT\}$; $\eta(k,u) \leftarrow curT$;
11 $\quad\quad V_{\mathcal{G}'} \leftarrow V_{\mathcal{G}'} \setminus \{u\}$; $k\text{-}prob(u, \mathcal{G}') \leftarrow -1$;
12 $\quad\quad \mathcal{A} \leftarrow \emptyset$;
13 $\quad\quad$ **for** $\forall (u,v) \in E_{\mathcal{G}'}$ **do**
14 $\quad\quad\quad$ *add* v *to* $\mathcal{A}$;
15 $\quad\quad$ *launch kernel: ComputeKProb*$(\mathcal{A}, k)$;
16 **return** *all* η-*thresholds.*

- ***adj***: An adjacency array of length $2|E_\mathcal{G}|$. For any index $i \in [p_s, p_e]$, $adj[i]$ stores a neighbor v of vertex u.
- ***prob***: A probability array of length $2|E_\mathcal{G}|$. For the same index $i \in [p_s, p_e]$, $prob[i]$ records the existence probability $p_{(u,v)}$ of the edge (u,v).

For example, Figure 3(a) shows the CSR representation of the uncertain graph $\mathcal{G}$ from Figure 1(a). For vertex v_1, since $off[v_1]$=1 and $off[v_2]$=5, its neighbors are stored in $adj[1..4]$, with corresponding edge probabilities in $prob[1..4]$.

Baseline. The sequential CTD (Algorithm 1) iteratively executes two key operations: (i) computing the k-probabilities of several vertices (lines 6 and 14), and (ii) identifying the vertex u with the minimum k-probability, denoted by min-k-probability, in the current active set (line 9). A naive GPU Baseline for η-threshold decomposition adopts this by: (i) allocating one thread per vertex for parallel k-probability computation, and (ii) utilizing a global variable, $global_{min}$, that all threads concurrently update via atomic operations upon finding a smaller k-probability. However, this one-thread-per-vertex mapping causes workload imbalance and warp divergence, while massive atomic updates to a single memory address trigger memory contention and execution serialization.

Gatd Framework. To address these challenges, we present the overall framework of Gatd (Algorithm 2). It begins by invoking a GPU-based core decomposition algorithm [1] to compute the core number $c(u)$ for each vertex $u \in V_\mathcal{G}$ and determine the maximum core value k_{max} (lines 1-2). For each k from k_{max} down to 1, Gatd restricts the uncertain graph to a subgraph $\mathcal{G}' = (V_{\mathcal{G}'}, E_{\mathcal{G}'}, p)$, containing all vertices with core number at least k. Then, the ComputeKProb kernel initializes the k-probabilities for all vertices in $V_{\mathcal{G}'}$ (lines 3-6). It next iteratively: (i) selects the min-k-probability vertex u via the ReduceMinKProb kernel, sets its η-threshold and removes u from $V_{\mathcal{G}'}$ (lines 9-11); and (ii) updates the k-probabilities of u's neighbors by re-invoking ComputeKProb (lines 12-15). This process continues until $V_{\mathcal{G}'}$ is empty.

3.2 Parallel k-Probability Computation

However, efficient parallel computation of k-probability faces two key challenges: (i) determining an appropriate smallest computing unit; and (ii) designing a suitable parallelization strategy based on the chosen computational unit to ensure minimal overhead.

The first challenge lies in selecting an appropriate parallel computation unit. The naive one-thread-per-vertex baseline suffers from uncoalesced memory accesses, whereas a one-block-per-vertex assignment can easily exhaust SM resources or induce severe load imbalance on large-scale graphs. To optimally navigate this trade-off, we propose a *warp-centric execution model*. Because the CSR format lays out vertex neighborhoods contiguously, explicitly assigning a single warp to process each vertex perfectly facilitates coalesced memory transactions, ensuring both robust execution efficiency and optimal hardware utilization.

The second challenge is to design an efficient intra-warp parallel strategy for computing k-probabilities. As outlined in Equation 3, the dynamic programming (DP) approach maintains a *state array* $X(h,j)$ for each vertex u, where $h \in [1, d(u)]$ is the number of processed incident edges and $j \in [0, k-1]$ represents the exact number of established neighbors. When processing edge e_{h+1}, all values $X(h+1, j)$ must be updated simultaneously based on $X(h,j)$. Because the update cost per j is uniform, we evenly distribute the k updates across the 32 threads of a warp. Specifically, thread i (where $i \in [0, 32)$) computes $X(h+1, i), X(h+1, i+32*1), X(h+1, i+32*2), \cdots$, for all items less than k. This assignment minimizes intra-warp divergence and enables fully parallel DP updates.

Algorithm 3 presents the ComputeKProb kernel for parallel k-probability computation. Each idle warp is allocated to a vertex $u \in V$ to collaboratively compute $k\text{-}prob(u, \mathcal{G}')$ (lines 1-2). The DP process begins by initializing the state arrays (line 4). For each neighbor nei of u in the current subgraph $\mathcal{G}'$ (lines 5-7), thread

Algorithm 3: Kernel Function ComputeKProb(V, k)

Input: A vertex set V and an integer k
Output: The k-probability of each vertex $u \in V$

1 **for** *warp* ***in parallel*** **do**
2 $u \leftarrow$ take an unprocessed vertex from V;
3 $p_s \leftarrow off[u]; p_e \leftarrow off[u+1] - 1$;
4 $X_1[0..k], X_2[0..k] \leftarrow 0.0; X_1[1] \leftarrow 1.0$;
5 **for** $i \in [p_s, p_e]$ **do**
6 $nei \leftarrow adj[i]; p \leftarrow prob[i]$;
7 **if** $k\text{-}prob(nei, \mathcal{G}') < 0.0$ **then** continue;
8 **for** $j \leftarrow laneId + 1; j \leq k; j \mathrel{+}= warpSize$ **do**
9 $X_2[j] \leftarrow p \cdot X_1[j-1] + (1.0 - p) \cdot X_1[j]$;
10 exchange the value of X_1 and X_2;
11 $S[laneId] \leftarrow 0.0$;
12 **for** $j \leftarrow laneId + 1; j \leq k; j \mathrel{+}= warpSize$ **do**
13 $S[laneId] \mathrel{+}= X_1[j]$;
14 **if** $laneId = 0$ **then**
15 $k\text{-}prob(u, \mathcal{G}') \leftarrow 1.0 - S[0]$;
16 $k\text{-}prob(u, \mathcal{G}') \mathrel{-}= S[j]$ for $\forall j \in [1, warpSize)$;
17 **return** *all computed k-probabilities.*

$laneId$ computes DP values at indices $(laneId + 1) + warpSize * n$ according to Equation 3 (lines 8-9). The arrays X_1 and X_2 are then swapped to propagate the DP state (Line 10). After processing all neighbors, thread $laneId$ aggregates its computed partial results into shared memory $S[laneId]$. Thread 0 reduces these values to compute the final $k\text{-}prob(u, \mathcal{G}')$ (lines 11-16). Finally, the kernel returns the computed k-probabilities for all vertices in V.

Example 3.1. *We illustrate the parallel k-probability computation using vertex v_5 from the uncertain graph in Figure 1(a), as depicted in Figure 3(b). Let $k = 3$, and assume each warp comprises 4 threads: $t_0, \ldots, t_3$. The first row of Figure 3(b) displays the initialized state array $X[0..3]$. The neighbors of v_5, namely $\{v_2, v_6, v_7\}$, are processed sequentially. When processing v_2, thread t_i for $i \in [0, 2]$ updates $X[i+1]$ in parallel. For instance, t_1 computes $X[2] = 1.0 * 0.8 + 0.0 * 0.2 = 0.8$. After all neighbors are processed, the final state array $X[0..3]$ is shown in the last row. The 3-probability of v_5 is then derived as: $3\text{-}prob(v_5, \mathcal{G}') = (1 - 0.056 - 0.332 - 0.468) = 0.144$.*

3.3 Min-k-probability Reduction

In the CTD algorithm, a min-priority queue maintains all remaining active vertices in $V_{\mathcal{G}'}$, each keyed by its current k-probability. During peeling, the vertex with min-k-probability is extracted and deleted from the queue. However, directly transferring this approach to GPUs is impractical for three reasons: (i) priority queue operations, e.g., extract-min/deletion, incur significant sequential overhead; (ii) global synchronization after each update undermines GPU parallelism; and (iii) irregular memory accesses break coalescing, degrading bandwidth utilization.

To address these issues, we employ a hierarchical parallel reduction strategy to identify the vertex with min-k-probability. Given an active vertex set V_{at}, it is uniformly partitioned among all threads. Each thread $t_{id} \in [0, threadNum)$ processes the subset:

$$V_{t_{id}} = \{u_j \in V_{at} \mid j \mod threadNum = t_{id}\} \tag{4}$$

Algorithm 4: Kernel Function ReduceMinKProb(V_{at})

Input: A vertex set V_{at}
Output: The vertex u_{glob} with minimum k-probability

1 assign vertex subset $V_{t_{id}} \subseteq V_{at}$ to thread t_{id} for processing;
2 $u_{t_{id}} \leftarrow \arg\min_{u \in V_{t_{id}}} k\text{-}prob(u, \mathcal{G}')$; // thread-level
3 **for** $\delta \leftarrow warpSize/2; \delta > 0; \delta \leftarrow \delta/2$ **do**
4 $u_{t_{id}} \leftarrow \arg\min_{u \in \{u_{t_{id}}, u_{t_{id}+\delta}\}} k\text{-}prob(u, \mathcal{G}')$;
5 **if** $laneId = 0$ **then** $minW[warpId] \leftarrow u_{t_{id}}$;
6 $\_\_syncthreads()$;
7 **if** $warpId = 0$ **then**
8 $u_{laneId} \leftarrow minW[laneId]$ **for** $\forall\ laneId < warpNum$;
9 **for** $\delta \leftarrow warpNum/2; \delta > 0; \delta \leftarrow \delta/2$ **do**
10 $u_{laneId} \leftarrow \arg\min_{u \in \{u_{laneId}, u_{laneId+\delta}\}} k\text{-}prob(u, \mathcal{G}')$;
11 $\_\_syncthreads()$;
12 **if** $threadId = 0$ **then**
13 $u_{glob} \leftarrow \arg\min_{u \in \{u_{glob}, u_{laneId}\}} k\text{-}prob(u, \mathcal{G}')$;
14 **return** u_{glob}.

Here, t_{id} and $threadNum$ denote a thread's global ID and the total number of threads, respectively. Then, the reduction proceeds in four levels: (i) thread-level: each thread t_{id} scans $V_{t_{id}}$ to find its local minimum vertex $u_{t_{id}}$; (ii) warp-level: threads within a warp cooperatively reduce their local results to obtain the per-warp minimum vertex; (iii) block-level: the per-warp outcomes are stored in shared memory, and the representative threads reduce the per-block minimum vertex; (iv) global-level: all per-block results are aggregated in global memory to yield the global minimum vertex u_{glob}. This design aligns with the GPU's hierarchical execution model, enabling high-throughput, low-overhead minimum selection.

Algorithm 4 provides the pseudocode for the ReduceMinKProb kernel. Specifically, lines 1-2 perform a thread-level scan to identify the local minimum vertex $u_{t_{id}}$ for each thread t_{id}. Then, lines 3-5 perform a warp-level reduction, recording the per-warp minimum vertex in the shared memory array $minW$. Next, lines 7-10 use threads in warp-0 of each block to reduce per-warp outcomes within it into the per-block minimum vertex. Finally, lines 12-14 aggregate the block-level minima in global memory to compute and return the global min-k-probability vertex u_{glob}.

Example 3.2. *We demonstrate Algorithm 4 using a configuration of 8 vertices, 2 thread blocks, 2 warps per block, and 2 threads per warp, totaling 8 threads: $t_0, \cdots, t_7$. Each thread is assigned one vertex, with its k-probability (shown in the first row of Figure 3(c)) serving as the local minimum. For clarity, we execute a trace for thread t_0. In warp-0 of block-0, t_0 compares its value (0.04) with that of t_1 (0.04) and records the warp-level minimum vertex v_0 in $minW[0]$. Then, t_0 reduces $minW[0..1]$ in block-0 to obtain the block-level minimum vertex v_0. Finally, the block-level minima from block-0 (0.04) and block-1 (0.258) are compared in global memory. Since $0.04 < 0.258$, v_0 is determined as the global min-k-probability vertex and returned.*

3.4 Complexity Analysis of Gatd

Although Gatd shares the same theoretical work complexity as the CPU-based CTD algorithm, it drastically reduces the actual execution time by mapping sequential bottlenecks to massively parallel GPU kernels:

(i) Parallel k-Probability Computation. The ComputeKProb kernel parallelizes the dynamic programming method across threads within a warp. This reduces the complexity of computing the k-probability for a vertex of degree d from $O(k \cdot d)$ to $O(\frac{k \cdot d}{warpSize})$.

(ii) Min-k-probability Reduction. The ReduceMinKProb kernel replaces the sequential min-priority queue with a hierarchical parallel reduction. Given $T = threadNum$ GPU threads, the time required to identify the global minimum k-probability vertex is reduced from $O(\log n)$ to $O(\frac{n}{T} + \log T)$, where $n = |V_G|$.

Thus, the overall parallel time complexity of Gatd is $O(k_{max} \cdot n \cdot (\frac{n}{T} + \log T) + k_{max}^2 \cdot m \cdot \frac{d_{max}}{warpSize})$, where $m = |E_{\mathcal{G}}|$ and d_{max} denotes the maximum degree. By transforming heavy per-vertex computations and global minimum searches into fine-grained parallel operations that align with the GPU's memory access architecture, Gatd achieves substantial practical speedups over CTD.

4 OPTIMIZATION TECHNIQUES

As highlighted in the Introduction, k-probability computation dominates the decomposition time, and dynamic, skewed workloads severely degrade GPU efficiency. To overcome these challenges, we introduce three key optimizations.

4.1 Redundancy Reduction

To minimize redundant k-probability computations, our first optimization introduces: (i) lower-bound pruning, which exploits the theoretical monotonicity of η-thresholds, and (ii) batch updating, ensuring each affected neighbor is updated once per iteration.

4.1.1 Lower-Bound Pruning. While lower-bound pruning has been used to accelerate core decomposition[1] on uncertain graphs [2, 8], existing techniques are ill-suited for our problem due to two main reasons. *(i) Divergent objectives.* Existing methods estimate a lower bound for a vertex's η-degree under a fixed parameter $\eta \in [0, 1]$. Conversely, η-threshold decomposition computes $\eta(k, u)$ (Problem 1), requiring us to bound the η-threshold for each vertex u across all possible $k \in [1, c(u)]$. Diverse problems necessitate distinct lower-bound criteria tailored to their specific objectives. *(ii) Prohibitive overhead.* Prior techniques typically employ complex estimators (e.g., the regularized beta function[2, 8]). While feasible for a single η query, repeatedly evaluating such functions across all k iterations introduces massive computational overhead and severe register pressure on the GPU's SIMT architecture.

To address this, we leverage the monotonicity of $\eta(k, u)$ with respect to k (Lemma 4.1) to establish a natural, computation-free, and safe lower bound. This strategy bypasses redundant probability updates for unpeelable vertices in current or near-future iterations, improving efficiency without incurring additional overhead.

Lemma 4.1. *Given an uncertain graph $\mathcal{G}$ and two integers k and k', for any vertex $u \in V_{\mathcal{G}}$, if $k < k'$, then $\eta(k, u) \geq \eta(k', u)$.*

proof. Assume $\eta(k, u) < \eta(k', u)$. According to Definition 2.4, there exists a subgraph $\mathcal{G}' \sqsubseteq \mathcal{G}$ containing u where every vertex v satisfies $k'\text{-}prob(v, \mathcal{G}') \geq \eta(k', u)$. However, by the monotonicity of k-probability (Definition 2.2), $k\text{-}prob(v, \mathcal{G}') \geq k'\text{-}prob(v, \mathcal{G}') \geq \eta(k', u) > \eta(k, u)$. Thus, u belongs to a (k, η')-core with $\eta' > \eta(k, u)$, contradicting the maximality of $\eta(k, u)$. □

Given that Gatd processes k in descending order from k_{max} to 1, Lemma 4.1 guarantees that the previously computed $\eta(k + 1, u)$ provides a natural, valid lower bound (lb) for $\eta(k, u)$. We exploit this property to design our lower-bound pruning technique. Specifically, during the k-th peeling iteration, we initialize lb for each vertex u in the active set $V_{\mathcal{G}'}$ (line 4) as follows:

$$lb(u) = \begin{cases} \eta(k+1, u), & \text{if } c(u) \geq k+1, \\ 0, & \text{if } c(u) = k. \end{cases} \tag{5}$$

Let $curT$ denote the current maximum η-threshold, i.e., the k-probability of the last peeled vertex. For any vertex u with $lb(u) > curT$, it follows that $\eta(k, u) > curT$. Hence, u cannot be peeled next and need not be added to the set $\mathcal{A}$ (line 14). Conversely, after kernel ReduceMinKProb returns the candidate vertex u_{cand} with the minimum observed k-probability (line 9), any vertex u with $lb(u) \leq k\text{-}prob(u_{candi})$ must have its k-probability recomputed to ensure global minimality.

4.1.2 Batch Update. Algorithm 2 iteratively removes the vertex with the minimum k-probability to assign its η-threshold. However, this strictly sequential approach is ill-suited for the parallel execution model of GPUs. Additionally, each vertex deletion triggers updates to the k-probabilities of its neighbors (lines 12-15). When multiple neighbors of a vertex are deleted successively, its k-probability is recomputed repeatedly, resulting in significant redundant computation.

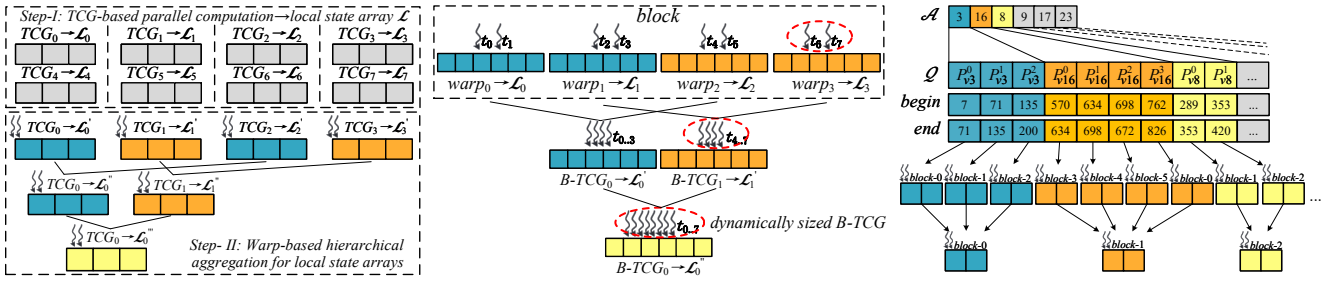
To alleviate this, we integrate the batch update into Algorithm 2. Specifically, in the current iteration, we identify all vertices that can be peeled simultaneously, falling into two cases: (i) vertices with the same minimum k-probability exceeding the current maximum η-threshold $curT$; these vertices are assigned the same η-threshold equal to this minimum k-probability. (ii) vertices with k-probabilities less than or equal to $curT$, which also share the same η-threshold, i.e., $curT$. Let Q denote the set of vertices from the both cases. We then construct an update set $\mathcal{A}'$ by collecting all active neighbors of vertices in Q (Algorithm 2, lines 12-14). Thus, each vertex in $\mathcal{A}'$ has its k-probability updated exactly once per iteration, thereby eliminating redundant recomputations.

Batching these updates enhances data-level parallelism and better aligns memory access and computation patterns with the GPU's parallel execution model. We thus integrate batch update into all enhanced algorithms, including those with adaptive parallelism and load balancing, while retaining the original sequential update only in the basic Gatd algorithm for fair comparison.

4.2 Adaptive Parallelism for Dynamic Workloads

To mitigate parallel inefficiencies caused by highly dynamic workloads across varying k values and iterations, our second optimization incorporates: **(i)** fine-grained warp parallelization via Thread Cooperative Groups to accommodate fluctuating k, and **(ii)** degree-aware hybrid scheduling to dynamically map execution units to iterative workloads.

[1] Given an uncertain graph $\mathcal{G}$ and a probability threshold $\eta \in [0, 1]$, core decomposition on uncertain graphs determines the *η-degree* for each vertex $v \in V_{\mathcal{G}}$, defined as the maximum integer k such that the probability of v having a degree of at least k is no less than η.



(a) Fine-grained warp parallelization (b) B-TCG-based hierarchical aggregation (c) Three-phase load balancing mechanism

Figure 4: Some Examples Illustrating our Optimizations

4.2.1 Fine-Grained Warp Parallelization. Gatd assigns one warp per vertex to compute its k-probability, with all 32 threads updating the dynamic programming (DP) state array $X(\cdot, j)$ for $j \in [0, k-1]$ in parallel. When $k \leq 16$, however, fewer than half the threads are active, causing severe intra-warp divergence. This motivates a tailored intra-warp execution model for small k values. For clarity, we first introduce the concept of thread cooperative group.

Definition 4.1 (***Thread Cooperative Group, TCG***). *Given an uncertain graph $\mathcal{G}$, an integer k, and a GPU warp of size $warpSize$ (typically 32), a thread cooperative group (TCG), is a contiguous subset of threads within the warp, whose size is dynamically determined by k. The number of threads per TCG, denoted N_{TCG}, is defined as $N_{TCG} = \min(2^{\lceil \log_2 k \rceil}, warpSize)$.*

Here, N_{TCG} is the smallest power of two no less than k, ensuring sufficient threads for parallel k-probability computation. A warp is thus partitioned into $\alpha = warpSize/N_{\text{TCG}}$ TCGs, denoted $\text{TCG}_0, \cdots, \text{TCG}_{\alpha-1}$. We now present our fine-grained TCG-based parallelization (FGWP) scheme, which mainly comprises two steps:

Step-I: *TCG-based parallel computation.* When processing a vertex u, its neighbor list is stored contiguously in CSR format over the index range $[off[u], off[u+1])$. Since each warp is divided into α TCGs, u's neighbors are evenly distributed among these TCGs:

$$V_{\text{TCG}_i} = \{id \in [off[u], off[u+1]) \mid (id - off[u]) \bmod \alpha = i\} \quad (6)$$

Each TCG_i independently computes a *local state array* $\mathcal{L}_i(j)$ for $j \in [0, k-1]$, where the j-th term represents the probability that u has a degree of j considering only the neighbors in V_{TCG_i}.

Step-II: *Hierarchical array aggregation.* To obtain the final state array $\tilde{\mathcal{L}}$ for vertex u, the α local arrays $\mathcal{L}_0, \cdots \mathcal{L}_{\alpha-1}$ are merged using a *hierarchical two-way strategy*. Specifically, in round $p (p \in [1, \log_2 \alpha])$, each pair of arrays $\mathcal{L}_i$ and $\mathcal{L}_{i+\alpha/2^p}$ (for $i \in [0, \alpha/2^p)$) is combined via Equation 7.

$$\mathcal{L}_i^{'}(j) = \sum_{q=0}^{j} \mathcal{L}_i(q) \cdot \mathcal{L}_{i+\alpha/2^p}(j-q), \forall j \in [0, k-1]. \quad (7)$$

This calculates the probability that u has a degree of j in the union of their respective neighbor subsets, i.e., $V_i^{'} = V_{\text{TCG}_i} \cup V_{\text{TCG}_{i+\alpha/2^p}}$. After each round, the number of local state arrays halves, from α to $\alpha/2$, and so on, until the final array $\tilde{\mathcal{L}}$ is obtained. The k-probability of u is then derived from $\tilde{\mathcal{L}}$ using Equation 2.

4.2.2 Hybrid Scheduling Granularity Selection. During each peeling iteration, Gatd updates the k-probabilities for the neighbors of currently deleted vertices (Algorithm 2, lines 12-15). These neighbors form a set $\mathcal{A}$, processed by the ComputeKProb kernel via a warp-centric assignment. However, $\mathcal{A}$ is typically small, often a few dozen vertices, even with the *batch update* optimization, leading to GPU underutilization. To resolve this, we propose a hybrid scheduling strategy that dynamically selects the parallelization granularity based on $\mathcal{A}$ and the current subgraph's average degree d_a:

- *Coarse-grained parallelization*: If $|\mathcal{A}| \leq blockNum$, assign one thread block per vertex.
- *Fine-grained parallelization*: If $|\mathcal{A}| > blockNum \cdot warpNum$, assign one warp per vertex.
- *Mixed-granularity parallelization*: Otherwise, split $\mathcal{A}$ by degree: **(i)** high-degree vertex set $\mathcal{A}_1 = \{u \in \mathcal{A} | deg(u) \geq d_a\}$ is processed coarsely; **(ii)** low-degree vertex set $\mathcal{A}_2 = \mathcal{A} \setminus \mathcal{A}_1$ is handled with fine-grained parallelization.

This degree-aware strategy allocates greater parallelism to computationally intensive vertices. Consequently, it supports two execution modes: warp-level (directly invoking ComputeKProb) and block-level (extending FGWP to a full block). For a vertex $u \in V_{\mathcal{G}}$, the block-level mode involves:

Step-I: *Warp-based parallel computation.* Neighbors of u are evenly distributed across warps within the block. Each warp computes a local state array $\mathcal{L}$ following Step-I of FGWP.

Step-II: *B-TCG-based hierarchical aggregation.* All local state arrays are merged to compute a final state array $\tilde{\mathcal{L}}$, akin to Step-II of FGWP, from which u's k-probability is derived.

While the original FGWP uses a fixed-size TCG to merge two local state arrays via Equation 7, this static configuration struggles under larger k values, where the increased merging cost exacerbates thread divergence and intra-block load imbalance. To mitigate this, we generalize TCG to the block level.

Definition 4.2 (***Block-level Thread Cooperative Group, B-TCG***). *Given a GPU block with $blockSize = warpNum \cdot warpSize$ threads, and α local state arrays to merge, let $N_m = \lceil a/2 \rceil$ denote the number of pairwise merging subtasks. A block-level thread cooperative group, denoted by B-TCG, is a contiguous subset of threads within the block, with size $N_{B\text{-}TCG} = blockSize/N_m$.*

Consequently, the block is partitioned into N_m B-TCGs. Each $\text{B-TCG}_i (i \in [0, N_m))$ merges arrays $\mathcal{L}_i$ and $\mathcal{L}_{2i}$.

4.3 Load Balancing for Skewed Workloads

In real-world power-law graphs [18], high-degree supernodes [38] incur orders of magnitude higher k-probability computation costs, causing severe load imbalance. To mitigate this, our third optimization proposes the three-phase collaborative load balancing (THLB) mechanism. THLB integrates **(i)** neighbor grouping, **(ii)** work stealing, and **(iii)** block-level aggregation to decompose prohibitive per-vertex workloads into balanced, fully parallelizable subtasks across streaming multiprocessors (SMs).

Phase-I: *Neighbor grouping.* For a high-degree vertex $u \in \mathcal{V}_{\mathcal{G}}$, we partition its neighbor list $N(u) = \{adj[i] | i \in [off[u], off[u+1])\}$ into multiple groups of approximately equal size. Each group is processed by an independent thread block (coarse-grained parallelization) to compute a local state array. Let N_s denote the partitioning granularity, i.e., target group size. This reduces the per-vertex computational complexity from $O(k \cdot d(u))$ to $O(k \cdot N_s)$, significantly alleviating the long-tail problem caused by supernodes.

The selection of N_s dictates a critical trade-off: a small N_s incurs excessive groups and higher scheduling cost, while a large N_s limits the efficacy of load balancing. We observed that the average degree of the refined graph $\mathcal{G}'$ (Algorithm 2, line 4) scales with k. To adaptively maintain optimal hardware occupancy, we set N_s as a linear function of k, i.e., $N_s = c \cdot k$, where $c = 32$ is an empirical constant. Consequently, $N(u)$ is divided into $\gamma = \lfloor d(u)/N_s \rfloor$ groups, denoted as $\mathcal{P}_u = \{P_u^0, \cdots, P_u^{\gamma-1}\}$. For each group P_u^i, its index range in *adj* array is recorded via *begin* and *end* arrays:

$$begin(P_u^i) = off[u] + i \cdot N_s, end(P_u^i) = \max\{begin(P_u^i) + N_s, off[u+1]\}. \tag{8}$$

Phase-II: *Work stealing.* Although Phase-I decomposes high-degree vertices into similarly sized neighbor groups, the per-vertex group number (γ) remains highly skewed. Static scheduling inevitably maps high-γ vertices to specific SMs, creating stragglers and severe inter-SM load imbalance. We mitigate this via dynamic work stealing. For the vertex set $\mathcal{A}$ to be processed in the current iteration, all neighbor groups generated in Phase-I are serialized into a global task pool Q in device memory. Each thread block, upon completing its assigned task, dynamically fetches a new unprocessed group from Q using atomic operations to ensure correctness.

Phase-III: *Block-level aggregation.* In the above two phases, the k-probability computation for a vertex u is decomposed into multiple subtasks, each processed independently by a thread block. Upon completion, each block writes its generated local state array to global memory. To obtain the final state array for u, from which its k-probability is derived, all local arrays must be aggregated. This is executed using the *B-TCG-based hierarchical aggregation* method introduced in the *Hybrid Scheduling Granularity Selection.*

Example 4.1. *We use Figure 6(d) to illustrate the execution flow of our THLB mechanism. Consider a GPU with 2 SMs, each containing 3 thread blocks, totaling 6 blocks labeled as block-*$0, \cdots,$ *block-*5*. Given* $k = 2$*, the grouping granularity is* $N_s = 32 * 2 = 64$*. For each vertex in the set* $\mathcal{A}$*, its neighbor list is partitioned into multiple groups of approximately* N_s *neighbors, which are enqueued into a global task pool* Q*. For example, vertex* v_3 *is split into three groups:* $\mathcal{P}_{v_3} = \{P_{v_3}^0, P_{v_3}^1, P_{v_3}^2\}$*, where* $P_{v_3}^0$ *covers neighbor indices* $[7, 71)$*, i.e.,* $begin(P_{v_3}^0) = 7$ *and* $end(P_{v_3}^0) = 71$*. During execution, idle blocks dynamically steal unprocessed groups from* Q*. For instance, block-*0 *may process* $P_{v_3}^0$*, while block-*1 *processes* $P_{v_3}^1$*, and so on. Each block computes a local state array for its assigned group. Once all groups are processed, local state arrays for vertex u are hierarchically aggregated via the B-TCG-based method to generate the final state array* $\tilde{\mathcal{L}}$*, from which* $k\text{-}prob(u)$ *is computed.*

Table 2: Detailed Descriptions of Uncertain Graphs

Type	Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	d_{max}	k_{max}
Real uncertain graphs	CNK	15,000	241,158	1,495	48
	Flickr	24,125	300,836	546	225
	DBLP	684,911	2,284,911	611	114
	Biomine	1,008,201	6,722,503	139,624	448
Semi-real uncertain graphs	Sup-User	192,409	714,570	14,294	61
	HepPh	28,093	3,148,447	4,909	410
	Wiki-Pe	1,409,666	2,267,114	1,095,589	120
	Stack	2,584,164	28,183,518	44,065	198
	Orkut	3,072,441	117,185,083	33,313	253
	Deli	38,289,740	137,240,382	1,057,753	180

4.4 Impact of Optimization Techniques

It is worth noting that while the time complexity of Gatd established in Section 3.4, i.e., $O(k_{max} \cdot n \cdot (\frac{n}{T} + \log T) + k_{max}^2 \cdot m \cdot \frac{d_{max}}{warpSize})$, provides a worst-case theoretical bound, our three optimization techniques drastically reduce the actual execution time:

(i) Redundancy Reduction (Section 4.1). Lower-bound pruning and batch updating effectively reduce unnecessary k-probability computations. Empirically, this scales the edge-related factor m down to a much smaller factor α ($\alpha \ll m$).

(ii) Adaptive Parallelism (Section 4.2). Fine-grained warp parallelization leverages TCGs to process disjoint neighbor subsets collaboratively, effectively reducing the degree-related factor d_{max}. Furthermore, the hybrid scheduling strategy applies a degree-aware split to dynamically assign high-degree vertices to full blocks, which expands the denominator from $warpSize$ to $blockSize$.

(iii) Load Balancing (Section 4.3). To mitigate the straggler effect caused by skewed degree distributions, our three-stage mechanism decomposes the d_{max} bottleneck into smaller group size N_s.

Together, these optimizations ensure high resource utilization and sustained scalability on massively skewed real-world graphs.

5 EXPERIMENTAL EVALUATION

We evaluate our algorithms using C++ implementations compiled with -O3 optimization. All experiments are conducted on a CentOS Linux 7 system with an Intel Xeon CPU (2.10 GHz) and an NVIDIA GeForce RTX 3090 GPU, which features 82 streaming multiprocessors (SMs) and 10,496 CUDA cores.

5.1 Experimental Settings

Datasets: We make use of four real uncertain graphs and six *semi-real* uncertain graphs to evaluate the performance of different algorithms. Table 2 provides the detailed statistical information of each dataset, where the last two columns show the maximum degree

Table 3: Runtime(sec) Comparison across Algorithms and Datasets

Algorithms	Real uncertain graphs				Semi-real uncertain graphs					
	CNK	Flickr	DBLP	Biomine	Sup-User	HepPh	Wiki-Pe	Stack	Orkut	Deli
CTD	115.62±0.88	244.89±0.33	90.49±1.43	INF	546.03±2.53	INF	INF	INF	INF	INF
Galois	36.79±1.07	34.54±0.77	185.20±1.38	INF	93.26±1.76	10,803.30±20.44	421.72±0.38	22,753.28±24.52	INF	INF
Julienne	14.92±0.19	19.06±0.21	120.30±1.13	9,883.51±26.07	65.07±0.45	5,403.10±12.74	332.35±1.98	INF	INF	INF
Gatd-OPT	**3.26**±0.02	**1.98**±0.02	**7.94**±0.06	**83.13**±0.85	**4.45**±0.02	**43.37**±0.13	**9.65**±0.04	**464.92**±9.94	**4,483.69**±13.76	**3,211.22**±12.61
Medusa	72.64±0.71	33.46±0.38	61.34±0.04	INF	769.92±1.73	12,429.34±14.35	22,258.27±62.76	INF	INF	INF
Gunrock	65.39±0.45	65.03±0.29	50.74±0.23	INF	447.57±2.00	10,187.64±6.45	9,280.64±29.34	INF	INF	INF

and the largest k. Notably, *semi-real* uncertain graphs are generated from deterministic networks by converting original edge weights w into probabilities using the function $f(w) = 1-e^{-w/2}$, following [7].

Algorithms: Since no dedicated multi-core CPU or GPU frameworks exist for η-threshold decomposition, we developed tuned baselines atop state-of-the-art infrastructures for a fair comparison. Multi-core CPU implementations utilize all 24 physical cores.

- **CTD**: A state-of-the-art single-threaded CPU baseline [44].
- **Galois** [29]: A multi-core method leveraging autonomously scheduled tasks to manage dynamic k-probability updates.
- **Julienne [10]:** A tuned multi-core framework utilizing bucketing primitives to accelerate min-k-probability search.
- **Medusa** [51] & **Gunrock** [40]: Generic GPU-based baselines employing vertex-centric and edge-centric programming models, respectively.
- **GatdBase:** A naive GPU baseline introduced in Section 3.1.
- **Gatd**: Our GPU-accelerated framework, integrating two core components: Pararrell k-probability computations (**-KP**) and Min-k-probability reduction (**-MR**).
- **Optimizations**. **-RC**: Reduces unnecessary k-probability computations. **-AP**: Employs adaptive parallelism for dynamic workloads. **-LB**: Improves load balancing for skewed workloads. **-OPT**: Integrates all above three optimizations.

Metrics: To ensure consistency, all computations utilize single-precision floating-point arithmetic, with outputs retained to six decimal places. We evaluate algorithmic efficiency based on the total decomposition time (in seconds). Due to runtime variability in GPU programs, each GPU-based implementation is executed 20 times, and both the average and standard deviation (std) are reported. For CPU-based algorithms, most of the experiments are repeated 5 times. If an algorithm fails to complete the decomposition within 8 hours, we denote it by INF.

5.2 Performance Comparison

We compare our optimized algorithm Gatd-OPT against existing approaches, with results shown in Table 3 (best performance in **Bold**, second-best underline). Overall, Gatd-OPT consistently outperforms all competitors across all datasets. Excluding timed-out instances, it delivers average speedups ranging from 46.0× (over Julienne) to 468.8× (over Medusa), with a peak speedup of 2,306.6× on Wiki-Pe. On large-scale datasets, e.g., Orkut and Deli, it is the only method to finish decomposition within the time limit. Our results yield two key architectural insights. First, Gatd-OPT substantially surpasses CPU-based methods (CTD, Galois, and Julienne), proving that massive GPU concurrency is essential to overcome the core-count limitation of multi-core CPUs. Second, generic GPU frameworks (Medusa, Gunrock) often underperform CPU-based baselines and fail on skewed datasets due to severe load imbalance and thread divergence under the strict SIMT model. This confirms that GPUs do not inherently guarantee performance for compute-intensive graph analytics without domain-specific co-design.

Table 4: Equivalence Test of Gatd against CTD (Evaluations use single-precision floating-point arithmetic with tolerance 10^{-6})

Dataset	EMR (%)	TMR (%)	AE	MAE
CNK	98.95%	99.46%	1.05e-08	1.01e-06
Flickr	99.80%	99.88%	2.05e-09	1.01e-06
DBLP	99.92%	99.94%	8.21e-10	1.01e-06
Sup-User	99.87%	99.93%	1.32e-09	1.01e-06

5.3 Equivalence Test

To validate Gatd's exactness against the CPU CTD, we conduct an equivalence test utilizing partial-order metrics rather than strict peeling-order matching, which fails due to valid tie-breaking of identical η-thresholds. For the four datasets where CTD completes, we report Exact Match Rate (EMR), Within-Tolerance Match Rate (TMR, with a tolerance of 10^{-6}), Average Absolute Error (AE), and Maximum Absolute Error (MAE), as detailed in Table 4. Specifically, Gatd demonstrates near-perfect equivalence, achieving an average EMR of 99.64%, with a peak of 99.92% on DBLP. Furthermore, TMR remains above 99.46%, while AE remains remarkably bounded from 8.21×10^{-10} to 1.05×10^{-8}. Although sparse MAE outliers occasionally shift the η-thresholds of marginal vertices, this boundary-case reordering remains confined to local fringe neighborhoods. Thus, Gatd efficiently preserves the core topology of the cohesive subgraphs, demonstrating robust algorithmic accuracy.

5.4 Scalability Evaluation

To evaluate scalability, we construct induced subgraphs by sampling 20% to 100% of the original vertices. Since CTD fails to complete on most datasets (Section 5.2), Figure 5 compares Gatd-OPT against multi-core and GPU baselines. We have three key observations. **(i)** Gatd-OPT consistently outperforms all competing methods across various datasets and scales, as is notably evident on Flickr, Biomine and HepPh. This is mainly attributed to its tailored parallelization strategies, ensuring high resource utilization. **(ii)** On challenging datasets such as Stack, Orkut, and Deli, competing methods begin to exceed the time limit beyond the 60% scale, while Gatd-OPT successfully processes the full graphs, underscoring its robustness and scalability. **(iii)** With increasing graph sizes, Gatd-OPT exhibits a noticeably flatter runtime growth curve than other methods (e.g., DBLP, Sup-User, and Wiki-Pe).

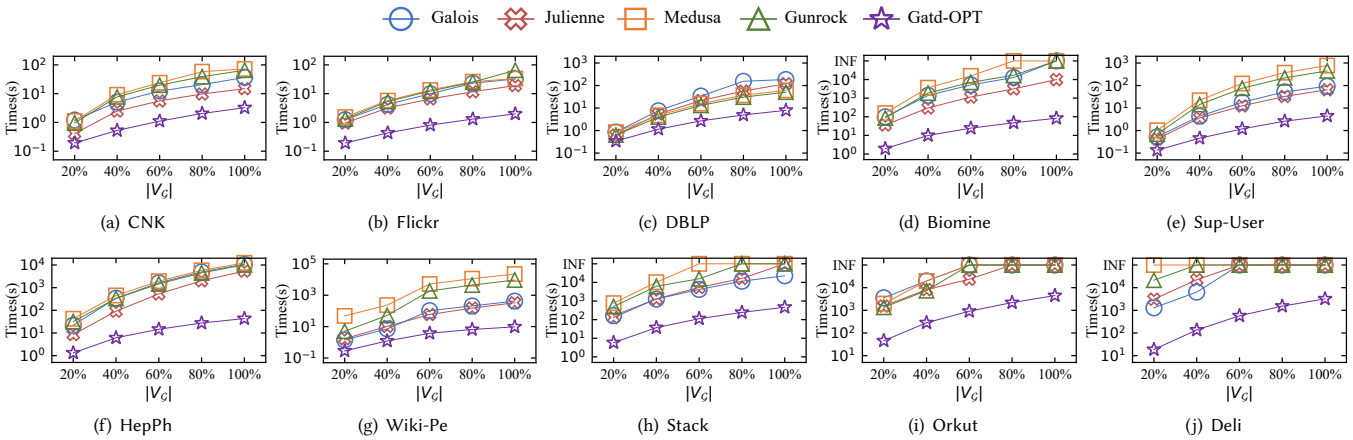


Figure 5: Scalability Evaluation

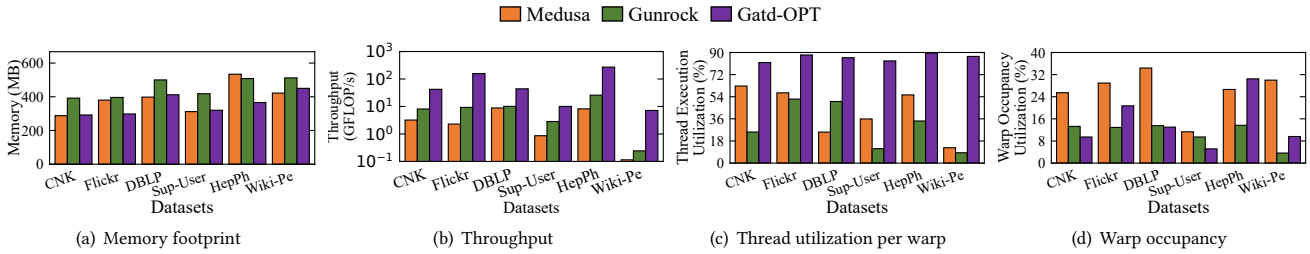


Figure 6: Analysis of Key Hardware Execution Metrics among GPU-based Methods

5.5 GPU Utilization Analysis

In this experiment, we report four critical hardware metrics: memory footprint, throughput, thread utilization per warp (a direct proxy for branch efficiency), and warp occupancy. Figure 6 compares Gatd-OPT against Medusa and Gunrock on datasets where all methods successfully complete, with results averaged over 10 profiling runs. Overall, Gatd-OPT exhibits superior hardware efficiency across all dimensions. First, it registers a relatively low memory footprint (e.g., on HepPh), demonstrating its scalability on massive graphs. Second, it consistently delivers the highest throughput and thread utilization; notably, the thread utilization on Wiki-Pe is 931.9× higher than that of Gunrock. Finally, Figure 6(d) reveals that Gatd-OPT pairs a moderate warp occupancy with maximized thread utilization. This yields a high instruction emission rate per warp, effectively masking execution latency without oversubscribing the SMs. Ultimately, these hardware metrics confirm that our framework effectively resolves the SIMT divergence bottlenecks, guaranteeing elevated GPU utilization.

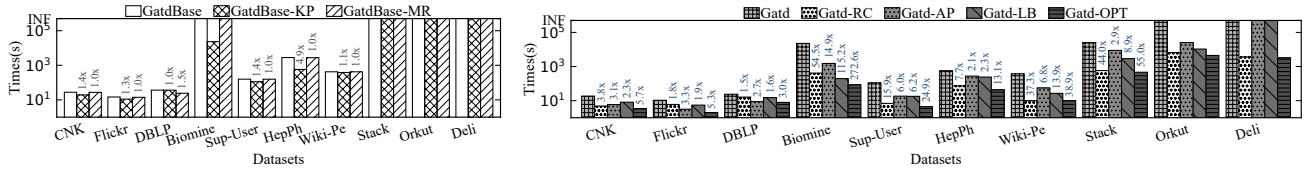
5.6 Ablation Study

This subsection evaluates our two core components (-KP, -MR) applied to the naive GatdBase, and three targeted optimizations (-RC, -AP, -LB) integrated into Gatd. Figure 7(a) reveals that parallel k-probability computation (-KP) is the primary acceleration engine, outperforming GatdBase and resolving timeouts (e.g., Biomine). Meanwhile, min-k-probability reduction (-MR) complements -KP by alleviating the inter-vertex synchronization overheads, yielding targeted speedups on datasets like DBLP. However, these foundational kernels yield limited relative gains in several cases because they cannot independently resolve intrinsic computational bottlenecks: redundant computations, severe SIMT divergence under dynamic workloads, and supernode-induced straggler effects.

To overcome these limitations, we evaluate our algorithmic optimizations in Figure 7(b). Redundancy Reduction (-RC) achieves the most significant performance improvement (averaging 20.8×) by pruning unnecessary computations via lower-bound pruning and batch updates. Adaptive Parallelism (-AP) enhances GPU utilization across varying k and iterations via a hybrid scheduling mechanism. It yields the highest relative gains on datasets with smaller maximum degrees (d_{max}), e.g., Flickr (3.3×) and DBLP (2.7×), where fine-grained warp parallelization is most advantageous. Lastly, Load Balancing (-LB) critically mitigates inefficiencies on structurally skewed graphs; notably, it delivers a 115.2× speedup on Biomine by systematically redistributing workloads from high-degree vertices. Together, these optimizations form Gatd-OPT, achieving an average overall speedup of 52.3× (peaking at 272.6× on Biomine) and conclusively validating our workload-aware GPU co-design.

5.7 Sensitivity Analysis

Finally, we investigate the sensitivity of parameter c in our three-stage load balancing strategy (Section 4.3), where the partitioning granularity N_s in Phase-I is defined as $N_s = c \cdot k$. Due to space constraints and trend consistency, Figure 8 presents representative results, demonstrating that varying c under fixed k reveals a



(a) Effect of Key Components

(b) Effect of Proposed Optimizations

Figure 7: Evaluation of Our Key Components and Optimizations

Figure 8: Effect of the Partition Parameter c

clear U-shaped runtime trajectory. Specifically, excessively large c values (e.g., $c = 2048$) degrade performance by creating oversized groups that exceed actual vertex degrees, nullifying load-balancing benefits and regressing to the baseline. Conversely, overly small c values (e.g., $c = 2$) induce severe task fragmentation, causing heightened overhead in task scheduling and inefficiency. Ultimately, $c = 32$ consistently strikes the optimal balance between mitigating load imbalance and minimizing task-scheduling overhead across all datasets, justifying its selection as our default parameter.

6 RELATED WORK

Cohesive subgraphs in uncertain graphs. Many cohesive subgraph models have been proposed for uncertain graphs, including core-based models (e.g., (k, η)-core [2], (α, β, η)-core [50], and (k, θ)-core [25, 31]), truss-based models (e.g., k-bitruss [52] and (k, γ)-truss [14, 16, 54]), and clique-based models (e.g., (k, τ)-clique [7, 20], α-clique [27, 34, 53], and (α, γ)-quasi-clique [26, 33]). In this work, we primarily focus on the (k, η)-core model.

Specifically, Bonchi et al. [2] first extended the k-core model to uncertain graphs by introducing the η-degree, computed via a dynamic-updating peeling algorithm. To accelerate queries, Yang et al. [44] and Wen et al. [41] developed the UCF-Index and its scalable extension UCEF-Index to precomputes η-thresholds. Further algorithmic improvements addressed the numerical inaccuracies of the original dynamic-updating method due to recursive floating-point number division [8] and enabled incremental maintenance for dynamic uncertain networks [6]. Despite these advances, η-threshold decomposition on large-scale uncertain graphs remains computationally prohibitive, necessitating more efficient algorithms.

GPU-accelerated core decomposition. Recent studies have leveraged GPUs to accelerate core decomposition. Zhang et al. [47] introduced degree-based peeling and locality property-based [30] iterative algorithms for temporal graphs. Mehrafsa et al. [24] reformulated the degree-based peeling algorithm using GPU vector primitives via PyTorch. Then, Ahmad et al. [1] developed a parallel queue-based strategy, yet its scalability on large graphs is constrained by memory bottlenecks. To address this, SpeedCore [49] compresses the search space and optimizes memory management through a cache-centric framework. Furthermore, numerous graph-parallel GPU systems have been developed to accelerate large-scale graph processing. Pioneering frameworks like Medusa [51] utilize a vertex-centric bulk synchronous parallel (BSP) model, whereas subsequent systems (e.g., CuSha [19], Gunrock [40], SGgraph [43]) have adopted edge-centric paradigms to further optimize execution.

Crucially, while η-threshold decomposition is a variant of core decomposition, its unique computational challenges render existing GPU-based methods ill-suited. First, classical core decomposition performs degree-based peeling with simple $O(1)$ degree updates. Correspondingly, the existing GPU-based frameworks (e.g., SpeedCore [49]) mainly focus on optimizing parallel queues and memory accesses. In contrast, η-threshold decomposition requires k-probability-based peeling, incurring costly $O(k \cdot d)$ probability updates. On GPUs, this intensive per-vertex workload exacerbates SIMT thread divergence and load imbalance, rendering conventional optimizations ineffective. Second, rather than a single peeling pass, η-threshold decomposition performs k-probability-based peeling across all possible $k \in [1, k_{max}]$ (Problem 1). Because the $O(k \cdot d)$ workload dynamically scales with k, the static thread allocation employed by existing frameworks fails to adapt. These fundamental distinctions strictly necessitate GPU parallel algorithms explicitly tailored for η-threshold decomposition.

7 CONCLUSION

This paper introduces Gatd, the first GPU-accelerated framework for η-threshold decomposition on uncertain graphs. For future work, we intend to explore η-threshold decomposition under correlated uncertain graph models and investigate Out-of-Core or distributed multi-GPU architectures for massive datasets.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62025206, U23A20296, and 62302444), and Zhejiang Province's "Lingyan" R&D Project under Grant No. 2024C01259. Yunjun Gao is the corresponding author of the work.

REFERENCES

[1] Akhlaque Ahmad, Lyuheng Yuan, Da Yan, Guimu Guo, Jieyang Chen, and Chengcui Zhang. 2023. Accelerating k-core decomposition by a GPU. In *ICDE*. 1818–1831.

[2] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *SIGKDD*. 1316–1325.

[3] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*. 233–244.

[4] Ziwei Chai, Yang Yang, Jiawang Dan, Sheng Tian, Changhua Meng, Weiqiang Wang, and Yifei Sun. 2023. Towards learning to discover money laundering sub-network in massive transaction network. In *AAAI*, Vol. 37. 14153–14160.

[5] Tianyi Chen and Charalampos Tsourakakis. 2022. Antibenford subgraphs: Unsupervised anomaly detection in financial networks. In *SIGKDD*. 2762–2770.

[6] Yu Chen, Qing Liu, Yifan Zhu, and Yunjun Gao. 2025. Efficient η-threshold maintenance in dynamic uncertain graphs. In *ICDE*. 515–528.

[7] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, Hongzhi Chen, and Guoren Wang. 2022. Fast maximal clique enumeration on uncertain graphs: A pivot-based approach. In *SIGMOD*. 2034–2047.

[8] Qiangqiang Dai, Rong-Hua Li, Guoren Wang, Rui Mao, Zhiwei Zhang, and Ye Yuan. 2023. Core decomposition on uncertain graphs revisited. *TKDE* 35, 1 (2023), 196–210.

[9] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *VLDB J.* 16, 4 (2007), 523–544.

[10] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*. 293–304.

[11] Marcus T Dittrich, Gunnar W Klau, Andreas Rosenwald, Thomas Dandekar, and Tobias Müller. 2008. Identifying functional modules in protein–protein interaction networks: an integrated exact approach. *Bioinformatics* 24, 13 (2008), 223–231.

[12] Yucheng Dong, Zhaogang Ding, Francisco Chiclana, and Enrique Herrera-Viedma. 2017. Dynamics of public opinions in an online and offline social network. *IEEE Trans. Big Data* 7, 4 (2017), 610–618.

[13] Jinyu Duan, Haicheng Guo, Fan Zhang, Kai Wang, Zhengping Qian, and Zhihong Tian. 2025. The k-trine cohesive subgraph and its efficient algorithms. In *SIGKDD*. 271–282.

[14] Fatemeh Esfahani, Jian Wu, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. 2019. Fast truss decomposition in large-scale probabilistic graphs. In *EDBT*. 722–725.

[15] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2021. Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions. In *SIGMOD*. 2829–2838.

[16] Xin Huang, Wei Lu, and Laks VS Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *SIGMOD*. 77–90.

[17] Xuanwen Huang, Yang Yang, Yang Wang, Chunping Wang, Zhisheng Zhang, Jiarong Xu, Lei Chen, and Michalis Vazirgiannis. 2022. Dgraph: A large-scale financial dataset for graph anomaly detection. *NeuralIPS* 35 (2022), 22765–22777.

[18] Sharad Jaiswal, Arnold L Rosenberg, and Don Towsley. 2004. Comparing the structure of power-law graphs and the Internet AS graph. In *ICNP*. 294–303.

[19] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC*. 239–252.

[20] Rong-Hua Li, Qiangqiang Dai, Guoren Wang, Zhong Ming, Lu Qin, and Jeffrey Xu Yu. 2019. Improved algorithms for maximal clique search in uncertain networks. In *ICDE*. 1178–1189.

[21] Shunyang Li, Kai Wang, Xuemin Lin, Wenjie Zhang, Yizhang He, and Long Yuan. 2024. Querying historical cohesive subgraphs over temporal bipartite graphs. In *ICDE*. 2503–2516.

[22] Xingyi Li, Wenkai Li, Min Zeng, Ruiqing Zheng, and Min Li. 2020. Network-based methods for predicting essential genes or proteins: a survey. *Brief. Bioinform.* 21, 2 (2020), 566–583.

[23] Guocheng Liao, Xu Chen, and Jianwei Huang. 2020. Privacy policy in online social network with targeted advertising business. In *INFOCOM*. 934–943.

[24] Amir Mehrafsa, Sean Chester, and Alex Thomo. 2020. Vectorising k-core decomposition for gpu acceleration. In *SSDBM*. 1–4.

[25] Xiaoye Miao, Yue Liu, Lu Chen, Yunjun Gao, and Jianwei Yin. 2022. Reliable community search on uncertain graphs. In *ICDE*. 1166–1179.

[26] Arko Provo Mukherjee, Pan Xu, and Srikanta Tirthapura. 2015. Mining maximal cliques from an uncertain graph. In *ICDE*. 243–254.

[27] Arko Provo Mukherjee, Pan Xu, and Srikanta Tirthapura. 2016. Enumeration of maximal cliques from an uncertain graph. *TKDE* 29, 3 (2016), 543–555.

[28] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM Rev.* 45, 2 (2003), 167–256.

[29] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[30] Lutz Oettershagen, Nils M Kriege, and Petra Mutzel. 2023. A higher-order temporal h-index for evolving networks. In *SIGKDD*. 1770–1782.

[31] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient probabilistic k-core computation on uncertain graphs. In *ICDE*. 1192–1203.

[32] Jose B Pereira-Leal, Anton J Enright, and Christos A Ouzounis. 2004. Detection of functional modules from protein interaction networks. *Proteins* 54, 1 (2004), 49–57.

[33] Lianpeng Qiao, Rong-Hua Li, Zhiwei Zhang, Ye Yuan, Guoren Wang, and Hongchao Qin. 2021. Maximal quasi-cliques mining in uncertain graphs. *IEEE Trans. on Big Data* 9, 1 (2021), 37–50.

[34] Ahmar Rashid, Muhammad Kamran, and Zahid Halim. 2019. A top down approach to enumerate α-maximal cliques in uncertain graphs. *J. Intell. Fuzzy Syst.* 36, 4 (2019), 3129–3141.

[35] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2021. Self-adaptive graph traversal on gpus. In *SIGMOD*. 1558–1570.

[36] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Real-time top-k personalized pagerank over large graphs on gpus. *PVLDB* 13, 1 (2019), 15–28.

[37] John A Stratton, Nasser Anssari, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Daniel Liu, and Wen-mei Hwu. 2012. Optimization and architecture effects on GPU computing workload performance. In *InPar*. 1–10.

[38] Xun Sun, Xi Xiao, Zhaoguo Wang, Guangwu Hu, Xuhui Jiang, Bin Zhang, and Hao Li. 2024. Understanding the influence of extremely high-degree nodes on graph anomaly detection. In *ICPR*. 19–34.

[39] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Shunyang Li. 2022. Discovering hierarchy of bipartite graphs with cohesive subgraphs. In *ICDE*. 2291–2305.

[40] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*. 1–12.

[41] Dong Wen, Bohua Yang, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2020. Computing k-cores in large uncertain graphs: An index-based optimal approach. *TKDE* 34, 7 (2020), 3126–3138.

[42] Jian Xu, Xiaoyi Fu, Yiming Wu, Ming Luo, Ming Xu, and Ning Zheng. 2020. Personalized top-n influential community search over large social networks. *World Wide Web* 23, 3 (2020), 2153–2184.

[43] Ala Eddine Yakhlef, Saïd Yahiaoui, and Ahcene Bendjoudi. 2025. SGgraph: A scalable GPU-based edge-centric graph processing framework. *Int. J. Parallel Program.* 53, 3 (2025), 18.

[44] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-based optimal algorithm for computing k-cores in large uncertain graphs. In *ICDE*. 64–75.

[45] De-Nian Yang, Hui-Ju Hung, Wang-Chien Lee, and Wei Chen. 2013. Maximizing acceptance probability for active friending in online social networks. In *SIGKDD*. 713–721.

[46] Ye Yuan, Lei Chen, and Guoren Wang. 2010. Efficiently answering probability threshold-based shortest path queries over uncertain graphs. In *DASFAA*. 155–170.

[47] Heng Zhang, Haibo Hou, Libo Zhang, Hongjun Zhang, and Yanjun Wu. 2017. Accelerating core decomposition in large temporal networks using gpus. In *ICONIP*. 893–903.

[48] Si Zhang, Dawei Zhou, Mehmet Yigit Yildirim, Scott Alcorn, Jingrui He, Hasan Davulcu, and Hanghang Tong. 2017. Hidden: hierarchical dense subgraph detection with application to financial fraud detection. In *SDM*. 570–578.

[49] Chen Zhao, Ting Yu, Zhigao Zheng, Yuanyuan Zhu, Song Jin, Bo Du, and Dacheng Tao. 2024. SpeedCore: Space-efficient and dependency-aware GPU parallel framework for core decomposition. In *ICPP*. 555–564.

[50] Gengda Zhao, Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Yizhang He. 2022. Efficient computation of cohesive subgraphs in uncertain bipartite graphs. In *ICDE*. 2333–2345.

[51] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *TPDS* 25, 6 (2013), 1543–1552.

[52] Alexander Zhou, Yue Wang, and Lei Chen. 2023. Butterfly counting and bitruss decomposition on uncertain bipartite graphs. *VLDB J.* 32, 5 (2023), 1013–1036.

[53] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *ICDE*. 649–652.

[54] Zhaonian Zou and Rong Zhu. 2017. Truss decomposition of uncertain graphs. *Knowl. Inf. Syst.* 50, 1 (2017), 197–230.