Efficient GPU-Accelerated Local Subgraph Counting

Qiao He
The Hong Kong Polytechnic University
lxqiao.he@connect.polyu.hk

Yiran Li*
University of Toronto
one.li@utoronto.ca

Man Lung Yiu
The Hong Kong Polytechnic University
csmlyiu@comp.polyu.edu.hk

Jieming Shi
The Hong Kong Polytechnic University
jieming.shi@polyu.edu.hk

ABSTRACT

Local subgraph counting computes the exact number of occurrences of a query graph around every vertex in a data graph. By capturing local higher-order structure, it supports extensive applications in network analysis and graph learning. The fastest existing method, SCOPE, accelerates counting through query graph decomposition, but it is designed for single-threaded CPU execution. As a result, it struggles on large graphs and cannot take advantage of modern GPU hardware. A naïve GPU adaptation is also ineffective: as the number of parallel GPU threads grows, the memory footprint of their intermediate results quickly drains the device memory.

We develop a high-performance GPU solution for local subgraph counting that preserves SCOPE's tree-decomposition framework while explicitly resolving the tension between massive GPU parallelism and limited device memory. Our approach compresses the intermediate join-and-aggregate results and proposes an insert-failure restart mechanism that guarantees correctness under bounded memory. We further design a key-mapping strategy that enables lock-free hash tables for higher throughput, eventually integrating these components into a complete GPU execution framework capable of handling arbitrarily complex queries. Experiments show that our GPU-accelerated method achieves up to a 35× speedup over a multi-threaded SCOPE implementation, reducing the processing time for a million-scale graph from days to about 20 minutes and making local subgraph counting practical at large scale.

PVLDB Reference Format:
Qiao He, Yiran Li, Man Lung Yiu, and Jieming Shi. Efficient GPU-Accelerated Local Subgraph Counting. PVLDB, 19(11): 3592 - 3605, 2026.
doi:10.14778/3836663.3836711

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lxhq/GPU-Local-Subgraph-Counting.

1 INTRODUCTION

Graphs model relationships in many domains, e.g., finance, social networks, and biology [6, 13, 35]. The problem of local subgraph counting [25, 43, 56] is to compute the number of occurrences of a query graph within a data graph around every vertex, and it serves as a fundamental tool for graph analysis. Formally, a local subgraph counting query is represented as $Q = (q, o)$, where q is a connected query graph and o is a vertex of q. Given a data graph G, a local subgraph counting algorithm computes, for each vertex v in G, how many matches of q map the vertex o in q to v.

Local subgraph counting has many applications in network analysis, such as summarizing network structures [41], graph clustering [54], and community detection [49]. In recent years, it has also gained significance in Graph Neural Networks (GNNs) [4, 25, 42], as incorporating local subgraph counting results into node embeddings has been shown to improve GNN performance by capturing higher-order local structures that manifest in large graphs.

However, local subgraph counting remains challenging because it requires enumerating subgraph isomorphisms and aggregating them at the designated vertex o. SCOPE is the state-of-the-art method [25] to solve this problem. It transforms the input query pattern into a set of smaller patterns and evaluates each small pattern by a tree-decomposition-based join-and-aggregation procedure. This design avoids enumerating all matches of the full query directly which substantially reduces the search space in practice. However, SCOPE is designed for single-threaded CPU execution, which limits its scalability and efficiency, especially for large and complex graphs. Furthermore, for GNN applications, their execution typically relies on powerful GPUs, but existing local subgraph counting algorithms have yet to utilize GPU parallelism, resulting in long computation times.

Exploiting GPU parallelism is therefore a natural direction. GPUs have accelerated related graph tasks, including subgraph matching [48, 52, 55], subgraph enumeration [16, 20], *global* subgraph counting [10, 53] and k-clique counting [2]. However, these methods cannot be directly applied to solve the local subgraph counting problem. Subgraph enumeration methods require materializing an exponential number of instances and incur a higher time complexity than counting methods. Global subgraph counting returns a graph-level pattern count. Even when these counts are computed from per-vertex or per-edge participation counts, those participation counts do not necessarily distinguish the structural role played by each data vertex in an occurrence. For local subgraph counting, each occurrence of a query $Q = (q, o)$ must contribute to the counter of the data vertex matched to o, which requires additional role-specific aggregation design.

In this paper, we propose a GPU solution for local subgraph counting that can leverage GPU's parallel processing capabilities. However, developing an efficient GPU implementation of SCOPE

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836711

Table 1: Frequently used notations

Symbol	Meaning
$G = (V_G, E_G)$	Data graph
$q = (V_q, E_q)$	Query graph
f	An isomorphism matching function
$\mathcal{F}$	A batch of isomorphism matching functions
T	A tree decomposition
τ	A tree node, which also represents a subgraph
$V(\tau)$	The set of vertices in τ
$V_{\text{share}}(\tau)$	Shared vertices between $V(\tau)$ and $V(\texttt{parent}(\tau))$
X_τ	A table aggregating matching results

introduces several challenges. SCOPE requires space complexity that grows linearly with the data graph. While this is acceptable for CPU single-thread execution, it becomes a severe scalability bottleneck on the GPU. A GPU contains thousands of threads, and fully utilizing its computational power requires launching a massive number of parallel tasks. Since each task demands memory proportional to the data graph size, the aggregate memory requirement quickly exceeds the limited capacity of GPU global memory.

Main contributions. Motivated by these challenges, (1) we employ compressed data structures for intermediate results, coupled with an insert-failure restart mechanism to guarantee correctness. This approach mitigates the linear dependency between per-task memory and the data graph size, allowing us to fit within GPU memory limits while maximizing parallelism. (2) We introduce a key mapping strategy that compresses variable-length hash-table keys into fixed 8-byte keys. This enables high-throughput *lock-free* GPU hash tables for any query graphs, overcoming the hardware constraints of atomic instructions. (3) We integrate these optimizations into a complete GPU solution that supports complex queries with arbitrary tree decompositions. (4) Experimental results demonstrate that our optimized GPU solution substantially outperforms both CPU and GPU baselines, achieving up to a 35× speedup over a multi-threaded SCOPE implementation. This acceleration transforms local subgraph counting from a multi-day workload into a minutes-scale operation, enabling its integration into GNN training and inference pipelines on million-vertex graphs.

2 PRELIMINARIES

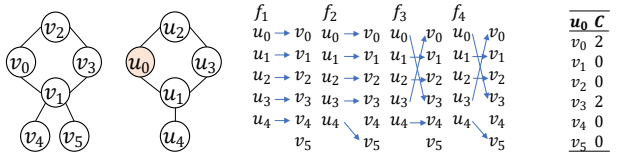
2.1 Basic Definitions

To maintain consistency with prior works [25, 39, 56], we study the local subgraph counting problem on undirected, unlabeled and connected graphs. We define the data graph as $G = (V_G, E_G)$, where V_G is the vertex set and E_G is the edge set, and query graph as $q = (V_q, E_q)$. A vertex in G is written as $v \in V_G$, and a vertex in q as $u \in V_q$. We write $(v, v') \in E_G$ for an edge in G and $(u, u') \in E_q$ for an edge in q.

Local subgraph counting is built upon the notion of *subgraph isomorphism*, defined as follows.

Definition 2.1 (Subgraph Isomorphism). Consider a data graph $G = (V_G, E_G)$ and a query graph $q = (V_q, E_q)$. A function $f : V_q \to V_G$ is a subgraph isomorphism if f is *injective* and for every edge $(u, u') \in E_q$, there exists a corresponding edge $(f(u), f(u')) \in E_G$.

Example 2.1. Consider the graph in Figure 1a as data graph G and the graph in Figure 1b as query graph q. We show all subgraph isomorphism functions from q to G in Figure 1c.



(a) G **(b)** q **(c) Isomorphism Matches** **(d)** LSC_{q,u_0}

Figure 1: An example of isomorphism match and local subgraph counting

We refer to a subgraph isomorphism function as a *match* and use $\text{ISO}(q, G)$ to denote the set of all matches from q to G.

Problem Statement (Local Subgraph Counting). A local subgraph counting query Q is defined as $Q = (q, o)$, where o is a vertex in q. For a vertex v in G, the local subgraph count, denoted $\text{LSC}_{q,o}(v)$, is the number of *matches* from q to G in which o is mapped to v:

$$\text{LSC}_{q,o}(v) = |\{f \in \text{ISO}(q, G) \mid f(o) = v\}|.$$

The local subgraph counting problem is to compute $\text{LSC}_{q,o}(v)$ for all $v \in V_G$, and we denote the resulting vector by $\text{LSC}_{q,o}$.

Example 2.2. Consider the graph in Figure 1a as the data graph G and the graph in Figure 1b as the query $Q = (q, u_0)$ for local subgraph counting. Figure 1c lists the matches from q to G, and the final result LSC_{q,u_0} is shown in Figure 1d. In matches f_1 and f_2, u_0 is mapped to v_0, giving $\text{LSC}_{q,u_0}(v_0) = |\{f_1, f_2\}| = 2$. Similarly, in f_3 and f_4, u_0 is mapped to v_3, yielding $\text{LSC}_{q,u_0}(v_3) = |\{f_3, f_4\}| = 2$.

Topologically equivalent query vertices yield identical local subgraph counts (e.g., u_0 and u_3 in Figure 1b) [43]. These vertices form *orbits*, which partition V_q into structurally equivalent sets [39]. For each orbit, it suffices to issue a single query $Q = (q, o)$, where o is any representative vertex. Some prior works [25, 56] apply a constant-time correction to account for duplicated matches arising from query symmetries, but this post-processing is independent of the data graph and orthogonal to our GPU acceleration focus, so we omit it.

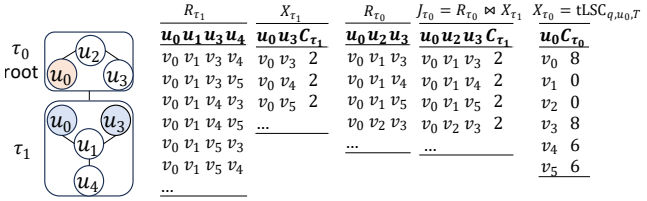
Frequently used notations are summarized in Table 1.

2.2 The SCOPE Method

SCOPE [25] is a recent high-performance method for local subgraph counting. It consists of two stages: the plan stage and the execution stage. The goal of the plan stage is to transform $Q = (q_{\text{in}}, o_{\text{in}})$ into a collection of simpler subproblems whose results can be combined to recover the original count. To this end, SCOPE constructs a set $\mathcal{Q} = \{(q_1, o_1, T_1, \mu_1), \ldots, (q_k, o_k, T_k, \mu_k)\}$, where each tuple contains a simplified query graph q_i, a designated vertex o_i, a tree decomposition T_i of q_i, and an integer weight μ_i. These weighted subproblems are designed so that their results combine exactly to produce the desired local subgraph counts $\text{LSC}_{q_{\text{in}},o_{\text{in}}}$. Given $\mathcal{Q}$ and a data graph G, SCOPE guarantees correctness through

$$\text{LSC}_{q_{\text{in}},o_{\text{in}}}(v) = \textstyle\sum_{(q,o,T,\mu)\in\mathcal{Q}} \mu \cdot \text{tLSC}_{q,o,T}(v), \tag{1}$$

where $\text{tLSC}_{q,o,T}$ denotes the intermediate tree decomposition-based count produced for (q, o, T) on G, without necessarily enforcing the injectivity constraint required by local subgraph counting. Because SCOPE's plan stage depends only on the query workload and is reused unchanged across data graphs, this paper focuses on



(a) A Tree Decomposition **(b) An example of tree decomposition-based local subgraph counting computation**

Figure 2: A tree decomposition and its tLSC result

accelerating the data-dependent execution stage, i.e., computing $\text{tLSC}_{q,o,T}$ for a given tree decomposition T and data graph G.

In the execution, $\text{tLSC}_{q,o,T}$, the tree decomposition-based local subgraph count, is computed using the query graph q and its tree decomposition T. We next introduce the concepts in this procedure.

Definition 2.2 (Tree Decomposition). Given a query graph $q = (V_q, E_q)$, a *tree decomposition* of q is a tree T in which each node $\tau \in T$ corresponds to a subset of vertices $V(\tau) \subseteq V_q$. A valid tree decomposition satisfies the following conditions: (1) for every vertex $u \in V_q$, there exists a tree node $\tau \in T$ such that $u \in V(\tau)$; (2) for every edge $(u, u') \in E_q$, there exists a tree node $\tau \in T$ such that $u, u' \in V(\tau)$; and (3) for every vertex $u \in V_q$, the set of tree nodes $\{\tau \in T \mid u \in V(\tau)\}$ forms a connected subtree of T.

Intuitively, a tree decomposition T partitions q into subgraphs represented by different tree nodes, ensuring that the structure of q is preserved by the three conditions defined. We define τ_{root} as the root of T, which could be any τ that contains the queried vertex o. We use $\texttt{parent}(\tau)$ and $\texttt{child}(\tau)$ to denote the parent and child nodes of τ, respectively. Furthermore, we define $V_{\text{share}}(\tau)$ as the set of vertices shared between $V(\tau)$ and $V(\texttt{parent}(\tau))$, i.e., $V_{\text{share}}(\tau) = V(\tau) \cap V(\texttt{parent}(\tau))$. We define $V_{\text{share}}(\tau_{\text{root}}) = \{o\}$.

Example 2.3. A tree decomposition of the query graph in Figure 1b (where $o = u_0$) is presented in Figure 2a. Since both τ_0 and τ_1 contain the u_0, either could serve as the root; we select τ_0 for this example. The resulting parent–child relationships are $\texttt{parent}(\tau_1) = \tau_0$ and $\texttt{child}(\tau_0) = \{\tau_1\}$. The shared vertices are identified as $V_{\text{share}}(\tau_1) = V(\tau_1) \cap V(\tau_0) = \{u_0, u_3\}$, and for the root, $V_{\text{share}}(\tau_0) = \{u_0\}$.

Given a tuple $(q, o, T, \mu) \in \mathcal{Q}$, SCOPE defines the tree decomposition based local subgraph counting, $\text{tLSC}_{q,o,T}$, as the result of following procedures.

The nodes $\tau \in T$ are processed in a *bottom-up* manner. For each node τ, all matches from τ to G are enumerated and stored in a relation R_τ. The attributes of R_τ correspond to $V(\tau)$, and its tuples represent the matched data vertices. For a leaf node τ, R_τ is aggregated into a relation X_τ on $V_{\text{share}}(\tau)$:

$$X_\tau(V_{\text{share}}(\tau), \text{C}_\tau) = {}_{V_{\text{share}}(\tau)}\Upsilon_{\texttt{count(*)}\to \text{C}_\tau}(R_\tau), \tag{2}$$

where ${}_g\Upsilon_{h\to A}$ denotes applying an aggregate function h for each group g, renaming the output of h as A.

For a non-leaf node τ, a relation J_τ is created by joining R_τ with all $X_{\tau'}$ for $\tau' \in \texttt{child}(\tau)$, where $\texttt{child}(\tau) = \{\tau_k, \ldots, \tau_w\}$. The shared vertex set $V_{\text{share}}(\tau')$ for each child τ' serves as the join key.

$$J_\tau(V(\tau), \text{C}_{\tau_k}, \ldots, \text{C}_{\tau_w}) = R_\tau \bowtie X_{\tau_k} \bowtie \cdots \bowtie X_{\tau_w} \tag{3}$$

J_τ is then aggregated into X_τ on $V_{\text{share}}(\tau)$ as:

$$X_\tau(V_{\text{share}}(\tau), \text{C}_\tau) = {}_{V_{\text{share}}(\tau)}\Upsilon_{\texttt{sum}(\text{C}_{\tau_k}\times\cdots\times\text{C}_{\tau_w})\to \text{C}_\tau}(J_\tau) \tag{4}$$

Finally, the relation $X_{\tau_{\text{root}}}$ represents the result of $\text{tLSC}_{q,o,T}$.

Example 2.4. Consider the data graph G in Figure 1a, the query graph q in Figure 1b (with $o = u_0$), and the tree decomposition T in Figure 2a. As established in Example 2.3, we have $\tau_{\text{root}} = \tau_0$, $\texttt{child}(\tau_0) = \{\tau_1\}$, $V_{\text{share}}(\tau_1) = \{u_0, u_3\}$, and $V_{\text{share}}(\tau_0) = \{u_0\}$. The computation process for $\text{tLSC}_{q,u_0,T}$, focusing on $u_0 \to v_0$, is illustrated in Figure 2b. First, the relation R_{τ_1} is computed, containing all matches from τ_1 to G. Since τ_1 is a leaf node, we aggregate R_{τ_1} into the table X_{τ_1} over the shared vertices $\{u_0, u_3\}$ according to Equation 2. Next, we compute R_{τ_0} (matches from τ_0 to G) and join it with X_{τ_1} using Equation 3 to obtain the intermediate relation J_{τ_0}. Finally, J_{τ_0} is aggregated into X_{τ_0} via Equation 4; the resulting relation X_{τ_0} represents the final counts $\text{tLSC}_{q,u_0,T}$.

SCOPE proves that the space complexity of the computation in Equation 2–Equation 4 is determined by the size of X_τ. In Example 2.4, to compute X_{τ_1}, we can generate matches in R_{τ_1} in a streaming manner: for each match, we immediately update the corresponding entry in X_{τ_1}. This allows us to perform the aggregation in Equation 2 without storing the entire relation R_{τ_1} in memory. Therefore, the only memory requirement is for X_{τ_1}, which holds the aggregated results. Similarly, for the join in Equation 3 and the aggregation in Equation 4, we can generate R_{τ_0} by one match at a time. After generating a match, we probe the relation X_{τ_1} and update the table X_{τ_0} accordingly. In practice, the relation J_{τ_0} is not explicitly materialized; it is computed dynamically during the join and aggregation. As a result, the memory usage is determined by the space required to store the X_τ relations at each tree node.

However, the space complexity of $X_\tau(V_{\text{share}}(\tau), \text{C}_\tau)$ grows exponentially with the number of shared vertices: $O(|V_G|^{|V_{\text{share}}(\tau)|})$. This can result in extremely large memory requirements. For example, for the data graph SY in Table 2, using 8-byte per count, maintaining $|V_G|^2$ counts would require 9,015 GB of memory.

To overcome this issue, SCOPE limits the size of X_τ by partitioning $V_{\text{share}}(\tau)$ into two *disjoint* parts:[1]

$$V_{\text{share}}(\tau) = V^{\text{prefix}}_{\text{share}}(\tau) \cup V^{\text{key}}_{\text{share}}(\tau)$$

Here, $V^{\text{key}}_{\text{share}}(\tau)$ is constructed to only contain *a vertex* or *an edge*. Intuitively, SCOPE splits the computation into iterations by fixing the prefix $V^{\text{prefix}}_{\text{share}}(\tau)$ to a specific match in each iteration. Then in each iteration, it executes the original bottom-up procedure (Equation 2–Equation 4) by extending from this fixed prefix match. Thus, aggregations are carried out only on the small key set $V^{\text{key}}_{\text{share}}(\tau)$. This ensures that X_τ depends on at most a vertex or an edge, reducing the space complexity to linear ($O(|V_G|)$ or $O(|E_G|)$).

Though many partitions are valid, shrinking $V^{\text{key}}_{\text{share}}(\tau)$ pushes more shared vertices into $V^{\text{prefix}}_{\text{share}}(\tau)$ and increases the number of prefix iterations. SCOPE proves that the time complexity for computing a node τ is exponential to the number of vertices in the prefix set of every tree node along the τ_{root}-to-τ path; hence the plan stage avoids an empty $V^{\text{key}}_{\text{share}}(\tau)$ and prefers an edge key when

[1] $V_{\text{share}}(\tau)$, $V^{\text{prefix}}_{\text{share}}(\tau)$, and $V^{\text{key}}_{\text{share}}(\tau)$ correspond to $V_C(\tau)$, $V^1_C(\tau)$, and $V^2_C(\tau)$ in [25].

available. In our implementation, we simply adopt the partition produced by the plan stage.

Example 2.5. Reconsider Example 2.4. The plan stage selects $V^{\text{prefix}}_{\text{share}}(\tau_1) = \{u_0\}$ and $V^{\text{key}}_{\text{share}}(\tau_1) = \{u_3\}$. An iteration is divided into 3 sequential steps (Figure 3c). Figure 3a illustrates the CPU execution of the execution stage. (1) A CPU Subgraph Enumeration algorithm (CPUSE) [47] is used to enumerate a single prefix match $f_0^P = \{u_0 \to v_0\}$. (2) SCOPE fixes f_0^P and follow the bottom-up order: the CPUSE extends f_0^P to generate matches for τ_1 one by one; the Join Aggregation Function (JAF) consumes each match to increment the corresponding slot in X_{τ_1} (an $|V_G|$-sized array), implementing the leaf aggregation defined in Equation 2. (3) Next, for the parent τ_0, the CPUSE extends f_0^P to generate matches one by one; for each match, the JAF probes X_{τ_1} to perform the join (Equation 3) and accumulates (Equation 4) the retrieved counts into X_{τ_0}. Finally, we reset X_{τ_1} and the CPUSE proceeds to enumerate the next prefix match f_1^P.

2.3 GPU Architecture

Graphics Processing Units (GPUs) have become increasingly popular for accelerating general-purpose computing tasks [16, 20, 48]. The GPU architecture consists of multiple Streaming Multiprocessors (SMs), each containing numerous cores that execute instructions in parallel. Using NVIDIA's CUDA framework [36], programs are implemented as kernel functions that run across parallel threads organized into blocks. Each block is assigned to an SM, and threads within a block are grouped into warps of 32 threads (referred to as lanes) that execute concurrently on the SM's cores.

The GPU memory hierarchy includes global memory, which is large but high-latency, accessible by all threads. Performance improves when threads access contiguous memory in a coalesced manner. Shared memory is low-latency on-chip memory, shared within a block, while registers provide the fastest access, storing variables private to each thread.

3 BASELINE GPU IMPLEMENTATION

A direct GPU adaptation of SCOPE's execution stage processes a batch of prefix matches concurrently in each iteration. In this baseline implementation, a GPU subgraph enumeration algorithm (GPUSE) [16] is used to generate a batch of prefix matches $\mathcal{F}^P$, which are then extended at each tree node.

To utilize the massive parallelism of a GPU, $|\mathcal{F}^P|$ must be sufficiently large. However, batching prefixes comes with a significant memory cost. As discussed in Section 2, when processing a single prefix match per iteration, the intermediate table X_τ has size proportional to the domain of its key: it is $O(|V_G|)$ when $V^{\text{key}}_{\text{share}}(\tau)$ is a single vertex and $O(|E_G|)$ when $V^{\text{key}}_{\text{share}}(\tau)$ is a single edge. With batching, we process a batch of prefix matches $\mathcal{F}^P$ per iteration, and thus must maintain a separate X_τ for each prefix match. As a result, the memory footprint becomes $O(|\mathcal{F}^P| \cdot |V_G|)$ or $O(|\mathcal{F}^P| \cdot |E_G|)$.

Suppose that we allocate the entire GPU device memory to the baseline implementation. Half of the device memory is used for maintaining X_τ tables while the other half is used for subgraph enumeration results and auxiliary data structures. We consider the vertex-key case, where $V^{\text{key}}_{\text{share}}(\tau)$ is a vertex as an example. For one fixed prefix match, the corresponding X_τ table has $|V_G|$ counters. Since each count uses 8 bytes, the aggregation memory required per prefix match is

$$m_{\text{array}} = 8\,\text{bytes} \cdot |V_G|. \tag{5}$$

Let $\mathcal{M}_\tau$ be the GPU memory budget, in bytes, for maintaining an X_τ table. The prefix batch size is bounded by

$$|\mathcal{F}^P|^{\text{array}}_{\max} = \left\lfloor \mathcal{M}_\tau / m_{\text{array}} \right\rfloor. \tag{6}$$

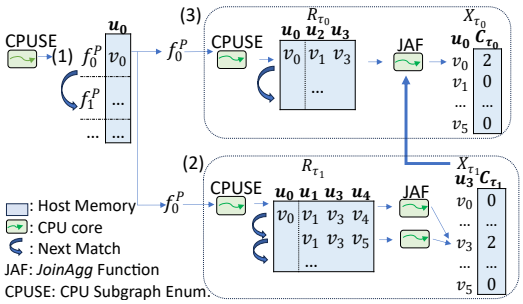
Example 3.1. Figure 3b illustrates the baseline GPU implementation corresponding to Example 2.5. (1) GPUSE generates a prefix batch $\mathcal{F}_0^P$ (size 2 for illustration). (2) For the leaf node τ_1, GPUSE extends $\mathcal{F}_0^P$ to generate matches, which are consumed by the JoinAgg kernel (JAK) to update X_{τ_1}. In this illustration, $|\mathcal{F}_0^P| = 2$: the two prefix matches map u_0 to two different data vertices, and for each match of u_0, u_3 can possibly be matched to any of the remaining $|V_G| - 1$ vertices. Thus, we need to allocate an array with $2 \cdot |V_G|$ slots for X_{τ_1}. (3) We similarly process the parent node τ_0. Finally, we clear intermediate tables and generate the next batch $\mathcal{F}_1^P$.

To quantify the scalability limit, consider graph SL (Table 2) on a GPU with 32 GB device memory, in which 16 GB is reserved for maintaining X_τ tables. In this baseline, table X_{τ_1} materializes a dense $|V_G|$-slot counter array *per prefix match* when $V^{\text{key}}_{\text{share}}(\tau_1)$ is a vertex. By Equation 5, with $|V_G| \approx 4.8 \times 10^6$, the per-prefix dense aggregation memory is $m_{\text{array}} = 8\,\text{bytes} \cdot |V_G| \approx 36\,\text{MB}$. Since 16 GB is reserved for X_τ tables, the prefix batch size is capped at roughly $16\,\text{GB}/36\,\text{MB} \approx \mathbf{455}$. For an edge key, the same analysis applies by replacing $|V_G|$ with $|E_G|$ in Equation 5.

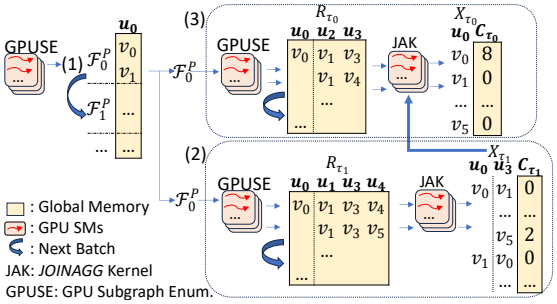
Example 3.1 illustrates the GPU memory–parallelism bottleneck: increasing the prefix batch size is necessary to expose GPU parallelism, but dense per-prefix aggregation arrays make the memory cost grow linearly with $|\mathcal{F}^P|$. Modern GPUs contain tens of thousands of hardware threads, and achieving high throughput requires launching enough parallel work to keep these threads busy and hide memory latency. This implies that $|\mathcal{F}^P|$ should be large; otherwise, most GPU cores remain idle. However, Example 3.1 shows that even in this minimal two-node setting, the baseline dense-array design allows at most **455** prefixes per iteration on SL under a 32 GB memory. This upper bound is already far below GPU-scale parallelism, and it becomes smaller for complex queries where multiple intermediate tables may coexist. This motivates our compressed intermediate representation and restart mechanism in Section 4.1.

4 GPU OPTIMIZATION TECHNIQUES FOR LOCAL SUBGRAPH COUNTING

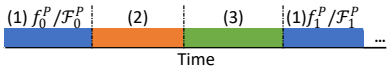
In this section, we introduce the optimizations that yield the most significant improvements in our experiments. To better illustrate these techniques, we first present a streamlined version on a *simplified yet representative* tree decomposition and then extend it to the general setting in the next section. Specifically, we consider a two-node tree decomposition with a root node τ_0 and a child node τ_1. We focus on the case where the plan stage selects a non-empty connected prefix set ($V^{\text{prefix}}_{\text{share}}(\tau_1)$) for τ_1 so that it can be enumerated directly by GPUSE. This is a minimal setting that exposes the GPU memory–parallelism tension targeted by our optimizations.



(a) An example of the execution stage



(b) An example of the basic GPU implementation of execution stage



(c) Execution schedule of Figure 3a and Figure 3b

Figure 3: Comparison of SCOPE and GPU-SCOPE

4.1 Compressed Intermediate Results with an Insert-Failure Restart Mechanism

As discussed in Section 3, the baseline GPU implementation allocates worst-case dense intermediate arrays for every prefix match in a batch, which severely limits the batch size and leaves the GPU underutilized. Example 3.1 shows that, for graph SL (Table 2), this worst-case allocation limits the prefix batch size to roughly **455**.

This subsection introduces our first optimization for alleviating this bottleneck: compressed intermediate storage with restart. The main idea is to store only the intermediate entries actually generated, rather than reserving space for all possible entries. As this compact storage is bounded by a fixed GPU memory budget, a large prefix batch may generate more entries than memory can accommodate. To handle this, our method safely processes the portion of the batch that fits and then restarts from the first unfinished prefix. This approach dynamically adapts the effective batch size to available memory while preserving the SCOPE counting procedure.

The opportunity for compression comes from the sparsity of the intermediate entries. Revisiting the running example in Example 3.1, for a fixed match of u_0, valid matches for u_3 are topologically constrained within the 2-hop neighborhood of $f(u_0)$. Given SL's average degree of 17.7, we expect approximately $17.7^2 \approx 313$ valid candidates for u_3 per prefix. This means the baseline allocates 4.8×10^6 slots to store only $\sim$ 313 values, yielding a utilization rate of merely 0.006%. Such sparsity is typical in practice, as most real-world data graphs are sparse [5, 8, 11]. To avoid this waste and enable larger prefix batches, we store intermediate results in X_τ using a compressed hash table and introduce an insert-failure restart mechanism to handle cases where the table becomes full.

Specifically, in each iteration, we enumerate a large batch of prefix matches $\mathcal{F}^P$ (see Example 4.1 for the estimation of this batch size) and maintain non-root X_τ as an *open-addressing* GPU hash table. Conceptually, X_τ is keyed by the match of $V_{\text{share}}(\tau)$, and we discuss how to encode this key efficiently in Section 4.2. Each key is stored in one of several candidate buckets determined by its hash value, and collisions are resolved by probing a sequence of buckets. We bound the maximum number of buckets probed per insertion by a fixed *probe budget*. If an insertion cannot find an available bucket within this probe budget, we say that the operation returns `insertFailed` and treat the table as full for the current iteration. The capacity of each table is a fixed value determined solely by the available GPU memory and is orders of magnitude *smaller* than the worst-case bound $|\mathcal{F}^P| \cdot |E_G|$ or $|\mathcal{F}^P| \cdot |V_G|$. We make no assumption that this capacity suffices to hold all counts; instead, insertion proceeds until an `insertFailed` indicates saturation. This design provides memory efficiency while retaining enough flexibility to support large prefix batches in practice.

For the compressed hash-table representation of X_τ, let K be the average number of distinct aggregation keys generated by one prefix, and let α be the effective load factor before insert failure. Each hash-table bucket stores an 8-byte key and an 8-byte value, so the bucket size is 16 bytes. The effective memory budget required per prefix is approximately

$$m_{\text{hash}} \approx (16\,\text{bytes} \cdot K)/\alpha. \tag{7}$$

Under the same X_τ-table memory budget $\mathcal{M}_\tau$, the supported prefix batch size is approximately

$$|\mathcal{F}^P|^{\text{hash}}_{\max} = \lfloor \mathcal{M}_\tau / m_{\text{hash}} \rfloor. \tag{8}$$

Example 4.1. Revisiting Example 3.1 on the SL dataset, our optimized implementation replaces the dense array X_{τ_1} with an open-addressing hash table. Empirically, with a probe budget of 64, we find `insertFailed` typically occurring at a load factor $\alpha \approx 0.4$. Using the estimate $K \approx 313$ distinct u_3 keys per prefix from above, Equation 7 gives $m_{\text{hash}} \approx 16\,\text{bytes} \times 313/0.4 \approx 12.2\,\text{KB}$. This is orders of magnitude smaller than the required aggregation memory 36 MB in Example 3.1. With $\mathcal{M}_\tau = 16\,\text{GB}$, Equation 8 gives a supported prefix batch size of roughly $\lfloor 16\,\text{GB}/12.2\,\text{KB} \rfloor \approx 1.3 \times 10^6$. Compared with the baseline upper bound of **455** prefixes (Example 3.1), this is a significant improvement.

When an insert-failure happens and `insertFailed` is raised, we handle it by recording a cut position (a variable called *cut*) and restarting from that cut point. We achieve this by giving each prefix match in $\mathcal{F}^P$ a unique prefix id. Specifically, we use the *IndexPrefixes* kernel in Algorithm 1 to augment every prefix match with an index attribute pid_τ, denoting the position of the prefix match in $\mathcal{F}^P$. We name the index by pid_τ because it identifies a fixed match of $V^{\text{prefix}}_{\text{share}}(\tau)$. GPUSE preserves this index when extending a prefix match, so every match consumed by JAK also carries the pid_τ value. For example, in Figure 4, we index each match of u_0 with pid_{τ_1} and this index also exists in both R_{τ_0} and R_{τ_1}.

Multiple threads may encounter `insertFailed` concurrently. We use `atomicMin`—an atomic primitive that updates a variable to

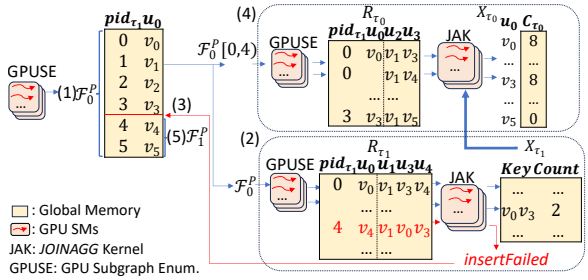


Figure 4: An example of execution stage's GPU implementation with an insert-failure restart mechanism

the minimum of its current value and the input value—to record the smallest failing prefix id. Concretely, *cut* is initialized to `UINT32_MAX` and if a thread processing a match f encounters `insertFailed`, `atomicMin`$(cut, f(pid_\tau))$ is performed. After the kernel returns, we truncate the prefix batch so that only prefixes with $pid_\tau < cut$ are treated as completed in the current iteration (i.e., we keep $\mathcal{F}^P[0, cut)$). Some GPU threads may continue executing after the first failure. This does not affect correctness because after truncation the parent probes X_τ only using matches extended from the completed prefixes.

By definition, $V_{\text{share}}(\tau_{\text{root}}) = \{o\}$, so $X_{\tau_{\text{root}}}$ always has space complexity $O(|V_G|)$. We therefore implement $X_{\tau_{\text{root}}}$ as a dense array of size $|V_G|$ rather than a hash table. This guarantees sufficient capacity at the root node for any batch size, and the `insertFailed` error can never occur at the root node.

Example 4.2. Reconsider the query and data graph used in Example 3.1. We implement X_{τ_1} as an open-addressing GPU hash table. The overall workflow with the insert-failure restart mechanism is shown in Figure 4. (1) We enumerate a batch of prefix matches $\mathcal{F}_0^P$ and augment it with *IndexPrefixes*$(\mathcal{F}_0^P, \tau_1)$. (2) We then extend $\mathcal{F}_0^P$ to matches of τ_1 by GPUSE. In the JAK, we assume the thread, which handles the match $\{pid_{\tau_1} \to 4, u_0 \to v_4, u_1 \to v_1, u_3 \to v_0, u_4 \to v_3\}$, encounters an `insertFailed` error. (3) We record 4 as the *cut* and cut $\mathcal{F}_0^P$ at the index 4, so that only $\{u_0 \to v_0\}, \cdots, \{u_0 \to v_3\}$ are treated as completed prefix matches in this iteration. (4) We extend the truncated set $\mathcal{F}_0^P[0, 4)$ from τ_0 to G and use JAK to do the join and aggregation. (5) In the next iteration, we clear the table X_{τ_1} and resume execution starting from $\{u_0 \to v_4\}$.

Termination safeguard. Theoretically, if `insertFailed` occurs while extending the *first unprocessed* prefix match in an iteration, then the iteration makes no progress and may lead to an infinite loop. For example, in Figure 4, the cut occurs at prefix $\{u_0 \to v_4\}$, so prefixes $\{u_0 \to v_0\}$ through $\{u_0 \to v_3\}$ are completed in that iteration. However, in a corner case, if `insertFailed` occurs while processing extensions of the first unprocessed prefix $\{u_0 \to v_0\}$, then the cut is at $cut = 0$, yielding zero completed prefixes. The next iteration restarts from the same prefix (i.e., $\{u_0 \to v_0\}$ in this example); if the same failure repeats (e.g., extensions of $\{u_0 \to v_0\}$ generate many distinct keys and, due to clustering—i.e., many keys hash to the same (or nearby) buckets—insertions may fail to find an available bucket within the fixed probe budget), the algorithm would keep cutting at the same position indefinitely. To guarantee termination, we detect this *no-progress* condition (an iteration completes *zero* prefix matches after a cut) and temporarily switch to the baseline GPU implementation (Section 3) with a *single-prefix* batch to process that prefix. Since the baseline uses dense arrays over the full key domain ($|V_G|$ or $|E_G|$ for vertex/edge keys), it never returns `insertFailed`, ensuring that at least one prefix completes. We then resume the hash-table-based execution for the remaining prefixes. Empirically, this safeguard is rarely triggered under a typical GPU memory budget. It occurs only in the hash-table size stress test under constrained memory budgets, where it guarantees execution completion while preserving exact counts.

Algorithm 1: Utilities for computing tree decomposition based local subgraph counting on GPU

```
1  Kernel IndexPrefixes(F^P, τ):
2      gtid ← global thread id;
3      f ← F^P[gtid];                          ▷ each thread picks a match
4      f(pid_τ) ← gtid;                        ▷ assign prefix id
5      F^P[gtid] ← f;                          ▷ write back the augmented match
6  Function MapKey(f, τ):
7      if |V^prefix_share(τ)| = 0 then
8          return getId(f(V^key_share(τ)));    ▷ root node
9      else
10         return f(pid_τ) ‖ getId(f(V^key_share(τ)))
```

4.2 Key Mapping for Lock-Free GPU Hash Tables

As discussed in the previous section, X_τ is implemented as an open-addressing hash table where keys represent matches of the shared vertex set $V_{\text{share}}(\tau)$. However, the size of $V_{\text{share}}(\tau)$ depends on the query graph q and its tree decomposition T. Consequently, the key length is variable and can easily exceed 8 bytes (e.g., a shared set of three vertices requires 12 bytes).

Variable-length keys are incompatible with high-throughput GPU hash tables. State-of-the-art GPU hash tables [3, 7, 19] maximize parallelism using *lock-free* designs that rely on hardware atomic instructions (e.g., `atomicCAS`) to claim or update buckets without explicit mutexes. On commodity GPUs, such atomic primitives are typically available only for fixed-width words (up to 8 bytes).[2] Implementing X_τ with variable-length keys would therefore force lock-based schemes (e.g., bucket mutexes), introducing severe contention and serialization.

To overcome this limitation, we propose a key mapping strategy that compresses any match of $V_{\text{share}}(\tau)$ into a fixed 8-byte integer. Recall that SCOPE partitions $V_{\text{share}}(\tau) = V^{\text{prefix}}_{\text{share}}(\tau) \cup V^{\text{key}}_{\text{share}}(\tau)$, where $V^{\text{key}}_{\text{share}}(\tau)$ contains exactly one vertex or one edge. Our mapping relies on two observations. (1) Within a prefix batch $\mathcal{F}^P$, each match of $V^{\text{prefix}}_{\text{share}}(\tau)$ can be uniquely represented by its position within the current prefix batch $\mathcal{F}^P$ and we already represent this by pid_τ, which can fit within a 4-byte integer. (2) The key part $V^{\text{key}}_{\text{share}}(\tau)$ is a single vertex or edge, which can be represented by a 4-byte vertex ID or edge ID.

Leveraging these observations, we encode the hash-table key of a match f as the concatenation of two 4-byte integers: Key =

[2] Some recent GPUs support wider atomics for certain operations, but the need for a *fixed-width* key remains: for complex queries, $|V_{\text{share}}(\tau)|$ can still exceed the supported atomic width.

Algorithm 2: Kernel $\mathcal{J}$oinAgg($\mathcal{F}, \tau, cut$)

Input: $\mathcal{F}$: a batch of matches, τ: a tree node, *cut*: the variable to record the smallest failing prefix id (initialized as UINT32_MAX)

▷ We use X_τ as a global variable

1 $gtid \leftarrow$ global thread id;
2 **if** $gtid \geq |\mathcal{F}|$ **then**
3 **return**;
4 $f \leftarrow \mathcal{F}[gtid]$;
5 **if** $|V^{prefix}_{share}(\tau)| > 0$ and $f(pid_\tau) \geq cut$ **then**
6 **return** ; ▷ early exit after any failure
7 **if** τ is a leaf node **then**
8 $cnt \leftarrow 1$;
9 **else**
10 $cnt \leftarrow \prod_{\tau' \in child(\tau)}$ read$(X_{\tau'}$, MapKey$(f, \tau'))$;
11 *success* $\leftarrow$ add$(X_\tau$, MapKey$(f, \tau), cnt)$;
12 **if not** *success* **then** ▷ insertFailed
13 atomicMin$(cut, f(pid_\tau))$;

$f(pid_\tau) \parallel$ getId$(f(V^{\text{key}}_{\text{share}}(\tau)))$, where $\parallel$ denotes bitwise concatenation. The helper function getId returns a unique 4-byte identifier: the vertex ID for vertex keys, or the CSR edge-array index for edge keys. Intuitively, $f(pid_\tau)$ uniquely identifies a fixed prefix match within the current batch and getId$(\cdot)$ uniquely identifies every vertex/edge. Therefore, the concatenation uniquely represents $f(V_{\text{share}}(\tau))$ within the current prefix batch/iteration, while remaining fixed-width. We implement this key mapping strategy as the function MapKey in Algorithm 1. The root node has an empty prefix set, so the key reduces to getId$(f(V^{\text{key}}_{\text{share}}(\tau)))$, which can be viewed as an 8-byte integer whose high 4 bytes are zero.

Example 4.3. Consider Figure 4. The bucket with key v_0v_3 in X_{τ_1} is created by the first match shown in R_{τ_1}, which maps $u_0 \rightarrow v_0$ and $u_3 \rightarrow v_3$. We refer to this match as f_0. Since f_0 is extended from the first prefix in $\mathcal{F}^P_0$, it carries $pid_{\tau_1} \rightarrow 0$. Since $V^{\text{key}}_{\text{share}}(\tau_1) = \{u_3\}$ and getId$(v_3) = 3$, applying MapKey gives

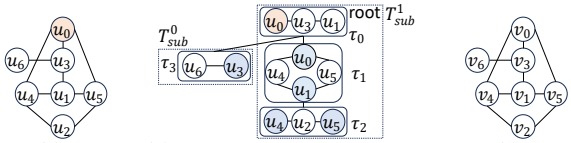
$$\text{MapKey}(f_0, \tau_1) = f_0(pid_{\tau_1}) \parallel \text{getId}(f_0(u_3)) = 0 \parallel 3.$$

Thus, the entry with key v_0v_3 in X_{τ_1} is stored internally using the 8-byte key $0 \parallel 3$.

Feasibility Analysis. This packing strategy requires that the batch size $|\mathcal{F}^P|$ and the graph size $(|V_G|, |E_G|)$ fit within 4-byte integers ($< 4.29 \times 10^9$). In the context of in-memory single-GPU processing, this constraint is **not a practical limitation**. Storing a graph with 2^{32} edges in a standard CSR format requires at least 16 GB of memory for the adjacency list alone. The memory demand increases significantly when accounting for the intermediate hash tables X_τ and the memory required to maintain the batch of subgraph enumeration results. Consequently, memory capacity becomes a bottleneck long before the 4-byte addressing limit is reached. Further, large-scale graph learning benchmarks fall comfortably within this limit (e.g., the largest graph in [17] contains 1.6×10^9 edges).

Unified Hash Table Interface. Although the underlying storage for X_τ differs—a dense array for the root node and an open-addressing hash table for other nodes—we describe them using a unified interface to simplify the algorithm description:

- boolean $\leftarrow$ add$(X_\tau, key, value)$
- $value \leftarrow$ read(X_τ, key)



(a) q **(b) Tree Decomposition** T **of** q **(c)** G

Figure 5: Examples of tree partitioning and a complex query

The function add atomically adds *value* to the entry associated with *key*. For dense arrays, this is a direct atomicAdd on the array index and always succeeds. For hash tables, add performs bounded-probe insertion/update; it returns false (insertFailed) if no bucket is found within the probe budget. The function read returns the current value for *key* (or zero if absent).

Join and Aggregate Kernel. Algorithm 2 shows the $\mathcal{J}$oinAgg kernel. *gtid* denotes the global thread id, which is also the match index in $\mathcal{F}$. Each thread processes one match f. Lines 5–6 implement an early exit. If a failure has already been recorded and f is extended from a prefix with $pid_\tau \geq cut$, the thread returns because this prefix will be deferred to the next iteration. Lines 7–10 compute the value *cnt* to be aggregated. If τ is a leaf node, $cnt = 1$. Otherwise, *cnt* is computed as the product (Equation 4) of values read from all child tables $X_{\tau'}$ using read and MapKey. Line 11 updates X_τ via add. For the root node, this maps to an atomic addition on the dense array and always succeeds. For other nodes, it attempts a bounded-probe update in the hash table. If the update fails, the thread atomically updates *cut* (Line 13), signaling the host to halt and restart.

5 GENERALIZING THE GPU OPTIMIZATIONS TO ARBITRARY TREE DECOMPOSITIONS

Section 4 presents our optimizations using a simplified two-node tree decomposition. This basic workflow is straightforward: the same prefix batch is extended first to build the child table, and then extended again at the root for probing. However, this workflow does not directly generalize to arbitrary tree decompositions. Certain nodes may lack an existing prefix batch to drive their execution, while different nodes may require different prefix batches.

We explain how to organize the computation for the general case. Our high-level idea has two parts. First, we divide the tree decomposition into smaller execution subtrees and process them bottom-up; this partitioning allows each subtree to be managed using a prefix-driven workflow similar to the simplified case. Second, instead of assuming a single global prefix batch for all nodes, the required prefix batches are generated locally within each subtree during execution. Below, we first illustrate why the simplified two-node setting is insufficient for the general case, and then describe the subtree-based execution for general tree decompositions.

Example 5.1. Figure 5a shows a query graph q with $o = u_0$ and its tree decomposition T in Figure 5b. We highlight the shared vertices in each tree node. In this example, the plan stage selects the following partition of each shared set:

- $V^{\text{prefix}}_{\text{share}}(\tau_0) = \emptyset$, $V^{\text{key}}_{\text{share}}(\tau_0) = \{u_0\}$
- $V^{\text{prefix}}_{\text{share}}(\tau_1) = \{u_0\}$, $V^{\text{key}}_{\text{share}}(\tau_1) = \{u_1\}$
- $V^{\text{prefix}}_{\text{share}}(\tau_2) = \{u_4\}$, $V^{\text{key}}_{\text{share}}(\tau_2) = \{u_5\}$
- $V^{\text{prefix}}_{\text{share}}(\tau_3) = \emptyset$, $V^{\text{key}}_{\text{share}}(\tau_3) = \{u_3\}$

This general structure violates two key assumptions from Section 4:

(1) *Empty prefix sets:* The leaf node τ_3 has $V^{\text{prefix}}_{\text{share}}(\tau_3) = \emptyset$. The streamlined routine in Section 4 assumes every non-root node is driven by extending a *non-empty* prefix batch. Therefore, τ_3 cannot be executed within the same prefix-driven batching loop and must be handled separately.

(2) *Divergent prefix sets:* Nodes τ_1 and τ_2 have different prefix sets ($\{u_0\}$ vs. $\{u_4\}$). In Section 4, we target a two-node case where only one non-root node introduces a prefix batch, so a single prefix-id attribute is sufficient for key mapping and restart. For general tree decompositions, different nodes can require batching over different prefix sets, which means prefix ids become *node-specific* and the prefix indexing strategy must be extended accordingly.

To enable efficient GPU execution for complex queries, we leverage SCOPE's *subtree partitioning* [25]. Given a tree decomposition T, the plan stage partitions it into a forest of execution subtrees $\mathcal{T} = \{T^0_{\text{sub}}, \ldots, T^n_{\text{sub}}\}$ by treating every *non-root* node τ with $V^{\text{prefix}}_{\text{share}}(\tau) = \emptyset$ as the root of a new subtree. This construction guarantees that within each subtree, every *non-root* node has a non-empty prefix set, and thus can be processed using prefix-driven batching. Since the partitioning is performed in the plan stage, we treat $\mathcal{T}$ as a known input in the execution stage. The plan stage also sorts $\mathcal{T}$ in a bottom-up order (i.e., bottom-up order on subtree roots in the original T). This schedule ensures that when we process a subtree, all dependent lower subtrees have already been fully computed and their root tables materialized, so higher subtrees can probe them during joins. For example, in Figure 5b, T is partitioned into T^0_{sub} (rooted at τ_3) and T^1_{sub} (rooted at τ_0). Since T^0_{sub} is executed first, when we later process τ_0 in T^1_{sub}, the join with τ_3 is handled by probing the precomputed table X_{τ_3}. Overall, subtree partitioning isolates empty-prefix nodes into standalone execution units and thus resolves the *empty-prefix* issue in Example 5.1. Note that subtree partitioning is a *logical* partition used for scheduling; it does not change the parent–child relationships in the original tree decomposition T. We still use $\texttt{child}(\tau)$ to denote all child nodes of τ in T and refer to the child nodes inside a subtree T_{sub} as $\texttt{child}(\tau) \cap T_{\text{sub}}$.

After partitioning, we categorize subtrees into two cases: a *trivial subtree* containing a single tree node, and a *complex subtree* containing multiple tree nodes. A trivial subtree can be handled by invoking GPUSE to enumerate matches and consuming them with the JAK. The main challenge lies in executing *complex subtrees* on a GPU, where the *divergent prefix sets* problem in Example 5.1(2) can arise. In the remainder of this section, we present how to execute a *complex subtree* on a GPU in Section 5.1 and then present the complete algorithm of our GPU solution in Section 5.2.

5.1 Executing a Complex Subtree

After subtree partitioning, each execution unit is a subtree T_{sub} whose internal non-root nodes all have non-empty prefix sets. Executing a complex subtree on a GPU presents a challenge: a middle node must be simultaneously compatible with its parent (so it can be probed under the parent-fixed prefix) and compatible with its children (so it can fix the prefixes required to probe them consistently); thus, a single prefix id is no longer sufficient.

Top-down Prefix Materialization in Batches. Recall that SCOPE partitions each shared set as $V_{\text{share}}(\tau) = V^{\text{prefix}}_{\text{share}}(\tau) \cup V^{\text{key}}_{\text{share}}(\tau)$, where $V^{\text{key}}_{\text{share}}(\tau)$ is exactly one vertex or one edge. Once a match of $V^{\text{prefix}}_{\text{share}}(\tau)$ is fixed, X_τ only needs to aggregate counts indexed by $V^{\text{key}}_{\text{share}}(\tau)$. This is why intermediate tables remain linear-size. It is also why the execution must ensure that a parent probes a child table under the same fixed prefix match used to build that table.

We follow SCOPE's incremental top-down strategy to materialize the prefix for each node, but implement it in a GPU-batched manner. Execution begins at the subtree root. For a node τ, we maintain a batch $\mathcal{F}$ of partial matches and extend it vertex-by-vertex using GPUSE according to the plan-stage order of $V(\tau)$, where all vertices in $V^{\text{prefix}}_{\text{share}}(\tau)$ appear first. Let u_{next} be the next query vertex in this order. GPUSE extends $\mathcal{F}$ with u_{next} and produces a new batch $\mathcal{F}_{\text{next}}$. This per-node enumeration follows the one-vertex-at-a-time matching in the style of worst-case optimal join (WCOJ) [34]: each step matches one new query vertex under constraints of previously matched neighbors, injectivity and ordering.

For every in-subtree child $\tau' \in \texttt{child}(\tau) \cap T_{\text{sub}}$, we *trigger* execution of τ' at the first step where $\mathcal{F}_{\text{next}}$ fixes all vertices in $V^{\text{prefix}}_{\text{share}}(\tau')$. Let $(u^{(0)}, \ldots, u^{(|V(\tau)|-1)})$ be the plan-stage order of $V(\tau)$. If we number enumeration steps by $\mathit{level} = 1, 2, \ldots, |V(\tau)|$, then τ' is triggered at the smallest *level* such that $V^{\text{prefix}}_{\text{share}}(\tau') \subseteq \{u^{(0)}, \ldots, u^{(\mathit{level}-1)}\}$. Once triggered, we execute τ' on a batch (or batch slice) derived from $\mathcal{F}_{\text{next}}$. After all triggered children have been executed, we continue extending τ's batch. After all vertices of $V(\tau)$ have been enumerated, we invoke JoinAgg to aggregate into X_τ by probing the tables of all child nodes. This batched top-down materialization differs from the CPU recursion in that each child is invoked on a batch of partial matches rather than a single partial match, exposing sufficient parallelism for GPU execution.

Prefix Indexing in a Complex Subtree. To support high throughput lock-free hash tables, we reuse the fixed-width key mapping strategy from Section 4.2. In a complex subtree, the key challenge is that different children τ' depend on different prefix sets; therefore, the unique prefix id $pid_{\tau'}$ for each child must be generated exactly when that child is triggered.

Our execution assigns $pid_{\tau'}$ at this trigger point. Specifically, when τ' becomes triggerable on a batch $\mathcal{F}_{\text{next}}$, we invoke IndexPrefixes on the batch (or its active slice) to assign the attribute $pid_{\tau'}$ to each match. GPUSE preserves this attribute during subsequent extensions, ensuring that all matches participating in the construction of $X_{\tau'}$ carry the correct $pid_{\tau'}$. Later, when the parent node τ probes $X_{\tau'}$ in JoinAgg, it applies MapKey to a match f extended from the same triggering batch and therefore carrying the same $pid_{\tau'}$, ensuring correct build–probe alignment.

Example 5.2. Consider the complex subtree T^1_{sub} in Figure 5b. We illustrate its GPU execution in Figure 6.

Step (1): enumerate τ_0 until τ_1's prefix vertices have been matched, then index the batch. We start from the subtree root τ_0. Once u_0 is matched, $V^{\text{prefix}}_{\text{share}}(\tau_1) = \{u_0\}$ is fixed and τ_1 becomes triggerable. We run IndexPrefixes on the current batch to assign each row a unique pid_{τ_1}. This produces the batch shown in Step (1), which we pass to τ_1 as $\mathcal{F}^P_{\tau_1}$.

Step (2): extend τ_1 until τ_2's prefix is matched, then index the batch. We execute τ_1 from $\mathcal{F}^P_{\tau_1}$ and extend the batch vertex-by-vertex using GPUSE. Since $V^{\text{prefix}}_{\text{share}}(\tau_2) = \{u_4\}$, τ_2 becomes triggerable when the current batch first matches u_4. At that point, we run

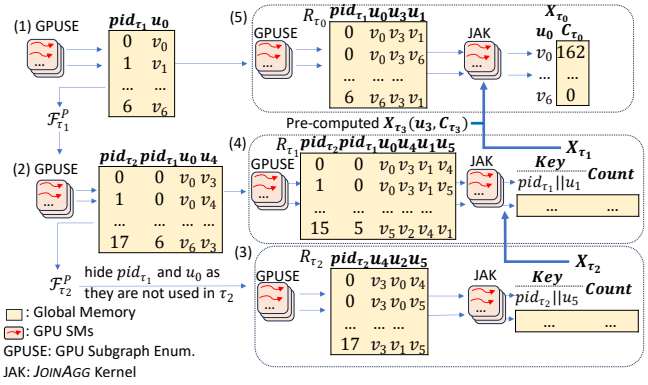


Figure 6: An example of processing a *complex subtree* on GPU.

IndexPrefixes to assign pid_{τ_2}. The resulting batch is shown in Step (2) and carries both pid_{τ_1} and pid_{τ_2}. We pass it to τ_2 as $\mathcal{F}^P_{\tau_2}$.

Step (3): execute τ_2 and build X_{τ_2}. Node τ_2 is a leaf in T^1_{sub}, so it has no in-subtree children to trigger. We therefore execute τ_2 directly from $\mathcal{F}^P_{\tau_2}$. For readability, we hide the inherited columns u_0 and pid_{τ_1}, since τ_2 does not use them. GPUSE generates R_{τ_2}, where each row is a match f that matches $V(\tau_2)$ to data vertices and carries pid_{τ_2}. JoinAgg aggregates R_{τ_2} into X_{τ_2}. For each $f \in R_{\tau_2}$, X_{τ_2} is keyed by $\texttt{MapKey}(f, \tau_2) = f(pid_{\tau_2}) \parallel \texttt{getId}(f(u_5))$.

Step (4): resume τ_1, probe X_{τ_2}, and build X_{τ_1}. After X_{τ_2} is materialized, we resume enumerating τ_1 to produce R_{τ_1}. For each match $f \in R_{\tau_1}$, JoinAgg probes X_{τ_2} using $\texttt{MapKey}(f, \tau_2)$ and aggregates into X_{τ_1} using $\texttt{MapKey}(f, \tau_1)$.

Step (5): resume τ_0, probe child tables, and produce X_{τ_0}. Finally, we resume enumerating τ_0 to produce R_{τ_0}. For each $f \in R_{\tau_0}$, JoinAgg probes X_{τ_1} and the precomputed table X_{τ_3} from subtree T^0_{sub}. The resulting products are aggregated into X_{τ_0}, which is the output table for this subtree.

5.2 The Complete GPU Solution

This subsection puts all pieces together and presents the complete GPU execution procedure for an arbitrary tree decomposition T. Before presenting the algorithm, we define the GPU subgraph enumeration(GPUSE) [16] API used in the algorithm.

- $iter \leftarrow \texttt{GPUSE}(\mathcal{F}, u_{\text{next}}, \tau, G)$
- $\texttt{boolean} \leftarrow iter.\texttt{hasNext}()$
- $\mathcal{F}_{\text{next}} \leftarrow iter.\texttt{next}()$

We use GPUSE as an iterator that extends a batch of partial matches $\mathcal{F}$ by one query vertex u_{next} at a time under the constraints of τ. Each call to $iter.\texttt{next}()$ returns a batch $\mathcal{F}_{\text{next}}$ whose size is bounded by a device-memory budget; $iter.\texttt{hasNext}()$ returns `false` only after all extensions have been produced. When $\mathcal{F} = \emptyset$, $iter.\texttt{next}()$ enumerates all data vertices as matches of u_{next}.

The complete algorithm is shown in Algorithm 3. In Lines 1–5, we allocate one table X_τ per tree node. If $|V^{\text{prefix}}_{\text{share}}(\tau)| = 0$, then τ is the root of an execution subtree and X_τ is implemented as a dense array. Otherwise, X_τ is an open-addressing hash table. In Lines 6–8, we execute each subtree $T_{\text{sub}} \in \mathcal{T}$. The plan stage orders $\mathcal{T}$ bottom-up so that any cross-subtree probe targets a table that has already been materialized. For each subtree, we initialize *cut* and invoke the host recursive routine $\texttt{helper}(\cdot)$.

In `helper`, *level* records how many vertices of $V(\tau)$ have been matched in $\mathcal{F}$. The variable *cut* is passed by reference so an insert-failure detected in any descendant can be observed and handled by its ancestors. In line 11, we first advance *level*. If $level > |V(\tau)|$, then all vertices in $V(\tau)$ have been enumerated and we invoke JoinAgg to update X_τ. Otherwise, in line 17, we create a GPUSE iterator to extend $\mathcal{F}$ into $\mathcal{F}_{\text{next}}$. The range $[start, end)$ is used to support truncation after a restart. In Figure 4, this range is $[0, 6)$ before the cut and becomes $[0, 4)$ after the cut. If $\mathcal{F}_{\text{next}}$ is truncated, the next

Algorithm 3: Tree decomposition based local subgraph counting on GPU

Input: T: the tree decomposition of (q, o), $G = (V_G, E_G)$
Output: $\text{tLSC}_{q,o,T}$

▷ $V_{\texttt{share}}(\tau)$, $V^{\texttt{key}}_{\texttt{share}}(\tau)$, $V^{\texttt{prefix}}_{\texttt{share}}(\tau)$, $\mathcal{T}$ are precomputed in the plan stage

1 **for** $\tau \in T$ **do**
2 **if** $|V^{\text{prefix}}_{\text{share}}(\tau)| = 0$ **then**
 ▷ $|X_\tau| = |V_G|$ if $V^{\texttt{key}}_{\texttt{share}}(\tau)$ is a vertex otherwise $|X_\tau| = |E_G|$
3 $X_\tau \leftarrow$ a dense array on GPU; ▷ global variable
4 **else**
5 $X_\tau \leftarrow$ a Hash Table on GPU; ▷ global variable
6 **for** $T_{\text{sub}} \in \mathcal{T}$ **do**
7 $\tau'_{\text{root}}, cut \leftarrow$ the root node of T_{sub}, UINT32_MAX;
8 $\texttt{helper}(\emptyset, \tau'_{\text{root}}, T_{\text{sub}}, G, 0, cut)$;
9 **return** $X_{\tau_{\text{root}}}$; ▷ τ_{root} is the root node of T
 ▷ cut is passed by reference
10 **Function** $\texttt{helper}(\mathcal{F}, \tau, T_{\text{sub}}, G, level, \texttt{ref } cut)$:
11 $level \leftarrow level + 1$;
12 **if** $level > |V(\tau)|$ **then**
13 JoinAgg$(\mathcal{F}, \tau, cut)$;
14 cudaDeviceSynchronize();
15 **return**
16 $u_{\text{next}} \leftarrow$ the $level$-th vertex in $V(\tau)$ in order;
 ▷ Extend $\mathcal{F}$ to contain u_{next} constrained by τ
17 $iter \leftarrow \texttt{GPUSE}(\mathcal{F}, u_{\text{next}}, \tau, G)$;
18 $start, end, \mathcal{F}_{\text{next}} \leftarrow 0, 0, \emptyset$;
19 **while** $iter$.hasNext() OR $end < |\mathcal{F}_{\text{next}}|$ **do**
20 **if** $end < |\mathcal{F}_{\text{next}}|$ **then**
21 $start \leftarrow end$;
22 **else**
23 $\mathcal{F}_{\text{next}} \leftarrow iter.\texttt{next}()$;
24 $start \leftarrow 0$;
25 $end \leftarrow |\mathcal{F}_{\text{next}}|$;
26 **for** $\tau' \in \texttt{child}(\tau) \cap T_{\text{sub}}$ **do**
27 **if** $level$ is the smallest number for $\mathcal{F}_{\text{next}}$ to contain $V^{\text{prefix}}_{\text{share}}(\tau')$ **then**
28 IndexPrefixes$(\mathcal{F}_{\text{next}}[start, end), \tau')$;
29 cudaDeviceSynchronize();
30 reset all buckets in $X_{\tau'}$ to empty;
31 $\mathcal{F}^P \leftarrow \mathcal{F}_{\text{next}}[start, end)$;
32 $\texttt{helper}(\mathcal{F}^P, \tau', T_{\text{sub}}, G, |V^{\text{prefix}}_{\text{share}}(\tau')|, cut)$;
33 **if** $cut \neq$ UINT32_MAX **then**
34 $end \leftarrow start + cut$;
35 $cut \leftarrow$ UINT32_MAX
36 $\texttt{helper}(\mathcal{F}_{\text{next}}[start, end), \tau, T_{\text{sub}}, G, level, cut)$;
37 **if** $cut \neq$ UINT32_MAX **then**
38 break;

Table 2: Statistics of the 14 data graphs used in experiments.

Graph	Notation	$\|V_G\|$	$\|E_G\|$	Avg. degree
web-spam	WS	4.8×10^3	3.7×10^4	15.7
rec-movielens-user-movies-10m	RM	7.6×10^3	5.5×10^4	14.6
bio-grid-yeast	BY	6.0×10^3	1.6×10^5	52.2
ca-astroph	CA	1.9×10^4	2.0×10^5	21.1
rec-github	RG	1.2×10^5	4.4×10^5	7.2
soc-gowalla	SG	2.0×10^5	9.5×10^5	9.7
soc-youtube	SY	1.1×10^6	3.0×10^6	5.3
web-BerkStan	WB	6.9×10^5	6.6×10^6	19.4
as-skitter	AS	1.7×10^6	1.1×10^7	13.1
web-wiki-ch-internal	WW	1.9×10^6	9.0×10^6	9.3
web-hudong	WH	2.0×10^6	1.4×10^7	14.6
ca-coauthors-dblp	CC	5.4×10^5	1.5×10^7	56.4
soc-livejournal1	SL	4.8×10^6	4.3×10^7	17.7
soc-orkut-dir	SO	3.1×10^6	1.2×10^8	76.3

loop iteration continues from *start* ← *end* (line 21). Otherwise it fetches a new batch from *iter*.`next()` (line 23).

Then, in line 26, we traverse all in-subtree children. If *level* is the smallest value such that the current partial match $\mathcal{F}_{\text{next}}$ contains $V_{\text{share}}^{\text{prefix}}(\tau')$, then τ' becomes triggerable: we index current slice, reset $X_{\tau'}$, and recurse into `helper`$(\mathcal{F}^P, \tau', \cdots)$ to materialize $X_{\tau'}$. In line 33, if the recursion reports an insert-failure by updating *cut*, we truncate the slice and reset *cut* so that the remaining matches can be retried in subsequent slices. After all children are handled for the current slice, we continue enumerating τ by recursively calling `helper` in line 36. Line 37—38 serves as an early exit if any insert-failure happened.

6 EXPERIMENTAL RESULTS

Baselines. We evaluate GPU-SCOPE-LF against three groups of methods: SCOPE-based CPU baselines, GPU-SCOPE ablation variants, and external GPU-based graph mining/counting baselines that can be adapted to local subgraph counting. All methods are executed from scratch without prebuilt indices to ensure a fair comparison of execution time.

- **SCOPE**: The original single-thread CPU implementation.
- **SCOPE-MT**: A multi-threaded CPU implementation that parallelizes independent prefix ranges and uses per-thread intermediate tables (20 threads in our runs).
- **GPU-SCOPE**: The baseline implementation that executes Algorithm 3 in Section 5 while maintaining each X_τ as a worst-case-sized dense array (Section 3).
- **GPU-SCOPE-LOCK**: GPU-SCOPE with compressed intermediate results using lock-based hash tables and the `insertFailed` restart mechanism (Section 4.1).
- **GPU-SCOPE-LF**: GPU-SCOPE-LOCK with key mapping and a lock-free hash table (Section 4.2).
- **VDMC-LSC**: VDMC [24] is a GPU-based local counting method designed for small patterns. We adapt it as VDMC-LSC to produce local subgraph counting results. It is enumeration-based and supports patterns up to 4 vertices.
- **G2Miner-LSC**: G2Miner [10] supports global counting for non-clique subgraphs up to 4 vertices using pattern-specific GPU kernels. We adapt it to produce local subgraph counts.

Hardware Settings. We run SCOPE and SCOPE-MT on Ubuntu 22.04 with an Intel i9-10900 2.80 GHz 20-thread CPU and 64 GB memory. All GPU variants run on an Ubuntu 22.04 server with an NVIDIA GeForce RTX 5090 GPU with 32 GB device memory and an AMD-9950X CPU. All C++/CUDA implementations are compiled with g++ 11.4.0 and CUDA 12.8. The SCOPE-MT algorithm runs with 20 CPU threads.

GPU Memory Budget. In the main experiments, each GPU-SCOPE variant allocates a fixed 30GB GPU memory budget, which is split between (i) the bounded matching-result buffer and (ii) aggregation storage: dense arrays for GPU-SCOPE or hash tables for GPU-SCOPE-LOCK/GPU-SCOPE-LF.

Datasets. We use 14 real-world graphs [25] from SNAP and Network Repository [23, 44] (Table 2). We convert each graph to a simple undirected graph by symmetrizing edges if needed, removing self-loops and multi-edges, extracting the largest connected component, and relabeling vertices to $\{0, \ldots, |V_G| - 1\}$.

Queries. We use *all* 58 5-vertex and *all* 407 6-vertex queries and, following SCOPE [25], randomly select 50 (out of 2,423) 7-vertex queries. Each query $Q = (q, o)$ is defined by a pattern q and an orbit representative o in q.

We have validated the correctness of our implementations by comparing the output counts to those of SCOPE. We report the average wall-clock execution time for each query group. For SCOPE-based methods, we exclude the plan stage because it is data-graph independent and implemented identically across these methods. For reference, SCOPE's plan stage takes on average around 0.07s for all 58 5-vertex queries, 7s for all 407 6-vertex queries, and 30s for the 50 7-vertex queries. For GPU methods, we stop timing only after all final counting results have been materialized in *CPU (host) memory*. We do not include disk I/O for writing outputs to persistent storage.

6.1 Performance Comparison

Figure 7 reports total execution times for the three query groups; full-height bars denote out-of-memory failures or runs exceeding the 20 h timeout. First, SCOPE-MT is excluded on the largest datasets (SL and SO) because it exceeds the 64 GB host-memory limit. It suffers from memory exhaustion on these large graphs because it maintains independent intermediate tables for 20 concurrent threads (requiring 20× the memory footprint of SCOPE). In addition, the GPU-SCOPE baseline is also omitted on SL and SO because it exceeds the 20 h timeout. As discussed in Section 3, GPU-SCOPE allocates worst-case-sized intermediate arrays, consuming $O(|\mathcal{F}^P| \cdot |V_G|)$ or $O(|\mathcal{F}^P| \cdot |E_G|)$ memory per iteration. On large graphs, the limited 32 GB device memory forces GPU-SCOPE to use an extremely small prefix batch $|\mathcal{F}^P|$, which severely underutilizes the GPU and necessitates a massive number of iterations.

This behavior is highlighted in Figure 8, where we report the average *prefix batch size* (the number of prefixes processed per iteration) for different pattern sizes. We can observe that GPU-SCOPE's average batch size drops sharply as graphs grow. In contrast, GPU-SCOPE-LOCK and GPU-SCOPE-LF sustain much larger batch sizes by using compressed hash tables, which helps explain why they maintain high GPU utilization on large graphs. These measurements are also consistent with our estimates in Example 3.1 and Example 4.1: on SL, GPU-SCOPE averages below 10^2 prefix per iteration, while GPU-SCOPE-LF exceeds 10^6.

5-vertex Queries. As shown in Figure 7, on the first 12 datasets where SCOPE-MT and GPU-SCOPE can finish, GPU-SCOPE-LF

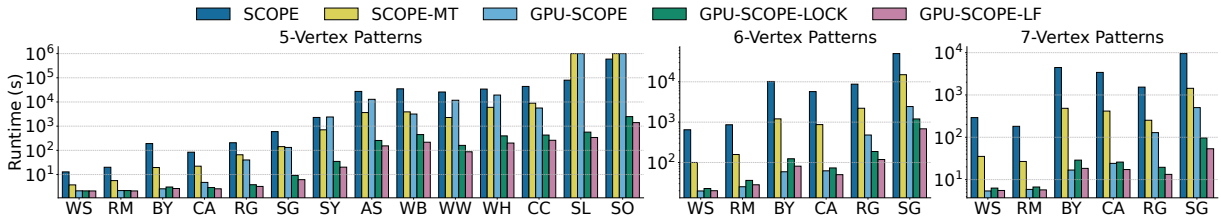


Figure 7: Total execution time of different methods

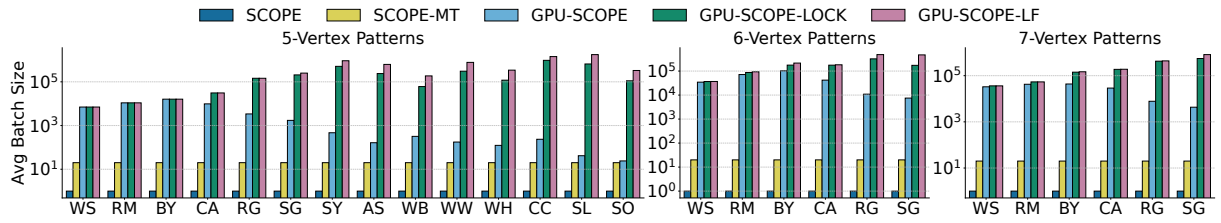


Figure 8: Average batch size of different methods

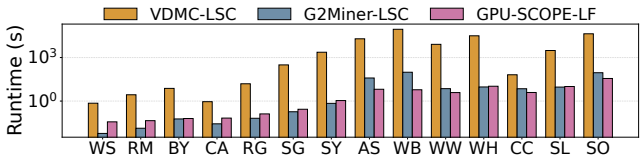


Figure 9: Runtime comparison of VDMC-LSC, G2Miner-LSC, and GPU-SCOPE-LF for all 11 4-vertex queries.

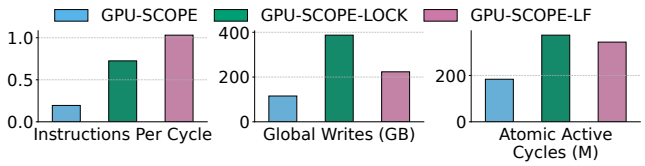


Figure 10: Micro-architectural Analysis of different variants

achieves an average 19.3× speedup over SCOPE-MT, with a maximum speedup of 35.0× on SY, and 42.5× speedup over GPU-SCOPE. We observe that GPU-SCOPE can be competitive on smaller graphs, but its efficiency severely deteriorates as the graph size grows, consistent with the limitation described in Section 3 and in Figure 8 (i.e., a small feasible $|\mathcal{F}^P|$ leads to severe underutilization of GPU).

6-vertex Queries. On the six smaller graphs (WS–SG) where all methods complete the full batch, GPU-SCOPE-LF achieves average speedups of 13× over SCOPE-MT and 2× over GPU-SCOPE.

7-vertex Queries. On the same six graphs, GPU-SCOPE-LF significantly outperforms all baselines, achieving average speedups of 17× over SCOPE-MT and 3× over GPU-SCOPE.

Overall, our GPU-SCOPE-LF reduces the execution time on million-scale graphs from hours- or days-level CPU workloads to minutes-level GPU computation. These gains come directly from our design choices, including compressed hash tables, failure-driven restarts, and large-batch prefix processing, which allow the GPU to be utilized far more fully than in baseline approaches. By tackling the memory–parallelism bottleneck, GPU-SCOPE-LF makes local subgraph counting practical for large-scale graph analysis, enabling feature extraction and downstream applications.

6.2 Comparison with External GPU Baselines

We compare with external GPU-based graph counting/mining baselines. Both VDMC-LSC and G2Miner-LSC rely on optimized GPU kernels for small-query workloads (up to 4-vertex queries), so they are not applicable to full 5-, 6-, and 7-vertex queries. Thus, in this experiment, the workload consists of *all* 11 4-vertex queries.

Figure 9 shows the comparison on all 11 4-vertex queries over all 14 datasets. On average, GPU-SCOPE-LF achieves a 1996× speedup over VDMC-LSC. Compared with G2Miner-LSC, which uses pattern-specific optimized kernels for 4-vertex motifs, GPU-SCOPE-LF remains competitive overall and is faster on several large datasets, while supporting general query patterns beyond 4 vertices.

6.3 Micro-architectural Analysis

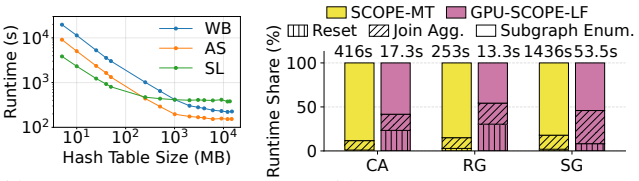
To reveal the execution characteristics on GPU, we profile each variant with NVIDIA Nsight Compute on SG using the 5-vertex workload, and report three micro-architectural metrics in Figure 10: (i) Instructions Per Cycle (IPC) [33], (ii) total global write traffic [32], and (iii) atomic-unit active cycles [14]. We compute IPC as a duration-weighted average across kernels, and sum global writes and atomic active cycles across all kernels. Higher IPC indicates fewer stalls (better latency hiding and utilization), while lower global write traffic and lower atomic active cycles indicate reduced DRAM pressure and reduced atomic serialization, respectively.

Compressed Intermediate Results. We first apply the optimization in Section 4.1 to GPU-SCOPE. GPU-SCOPE-LOCK processes much larger prefix batches under the same GPU memory budget, reducing kernel launch overhead and exposing more parallel work. As shown in Figure 10, this substantially increases IPC. At the same time, GPU-SCOPE-LOCK shows higher global write traffic and atomic active cycles. This is expected because hash-table insertion/update incurs extra random memory updates and synchronization, which are not present in the simple dense-array writes. Importantly, this additional per-update overhead is outweighed by the improved IPC, leading to the observed end-to-end speedup.

Lock-Free Hash Table. We evaluate Key Mapping optimization in Section 4.2. GPU-SCOPE-LOCK uses spinlocks to update hash-table buckets, causing serialization under contention. GPU-SCOPE-LF maps keys to fixed 8-byte integers and uses hardware atomic CAS

Table 3: Restart profiling of GPU-SCOPE-LF on representative 6-vertex workloads. (HT Occ. = Hash-table Occupancy)

Dataset	HT Occ.	# Restarts	Total Time (s)	Restart Time (s)
CA	1.82%	8	52.48	0.81
BY	6.30%	52	81.94	2.64
RG	13.67%	149	119.88	4.74
SG	43.24%	1605	667.84	15.96



(a) Hash-table size stress. (b) Runtime breakdown.

Figure 11: Hash-table size stress and runtime breakdown.

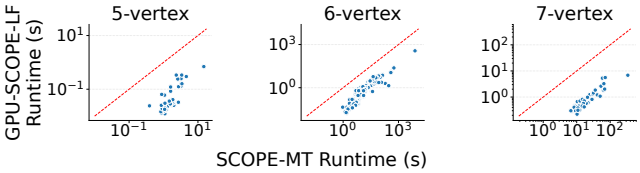


Figure 12: Single-pattern runtime comparison on SG.

for lock-free updates. In Figure 10, GPU-SCOPE-LF further improves IPC and reduces global write traffic and atomic active cycles, confirming that key mapping reduces synchronization overhead.

6.4 Profiling and Detailed Runtime Analysis

Restart statistics and hash-table occupancy. Table 3 reports the hash-table occupancy and the restart statistics of GPU-SCOPE-LF on representative 6-vertex workloads. While higher hash-table utilization increases restart frequency, the runtime overhead caused by restarts remains low (i.e., at most 3.96% of the total runtime). This validates that the restart mechanism enables bounded-memory execution without degrading the end-to-end performance.

Hash-table size stress. We stress GPU-SCOPE-LF by varying the hash-table memory budget in Figure 11a. Reducing the hash-table budget would restrict the effective batch size and lead to more frequent restarts. Under the typical memory budget used in our experiments, GPU-SCOPE-LF remains effective.

Single-pattern runtime. To verify that the overall speedup is not driven by a small subset of query patterns, we compare SCOPE-MT and GPU-SCOPE-LF on individual query patterns. In Figure 12, each point represents one query pattern; the x-axis and the y-axis report the runtime of SCOPE-MT and GPU-SCOPE-LF, respectively. GPU-SCOPE-LF is consistently faster across different query patterns.

Runtime breakdown. Figure 11b decomposes total runtime into subgraph enumeration, join aggregation, and table reset. GPU-SCOPE-LF accelerates all three, with the largest improvement in subgraph enumeration. Join aggregation and table reset show smaller speedups because: (i) hash-table aggregation introduces extra memory accesses and atomic updates; (ii) SCOPE-MT resets only touch dense-array entries while GPU-SCOPE-LF conservatively clears all allocated hash-table buckets.

7 RELATED WORK

Subgraph Counting. Subgraph counting methods include matrix-, enumeration-, and decomposition-based approaches [43]. Matrix-based methods derive counts using algebraic formulations [29–31]. Enumeration-based methods explicitly enumerate matches [37, 51]; VDMC [24] is one such GPU-based local counting method, relying on optimized GPU kernels for small patterns (up to 4-vertex queries). Decomposition-based methods, including ESCAPE [40], DISC [56], and SCOPE [25], decompose queries into smaller patterns; DISC and SCOPE support arbitrary queries. To the best of our knowledge, there is no prior GPU-accelerated local subgraph counting method that supports arbitrary query patterns. Global graph mining/counting systems [9, 10, 28] are related, but adapting them to local counting is non-trivial because local counting requires orbit-aware per-vertex aggregation rather than only graph-level or role-agnostic vertex/edge-centric counts. Approximate methods use learning-based estimators [21, 46, 50, 57] or sampling [22, 38, 45, 53].

GPU Subgraph Enumeration. GPU subgraph enumeration methods are mainly BFS-based [15, 16, 27] or DFS-based [12, 20]. BFS-based methods materialize partial-match batches and extend them one query vertex at a time, which is aligned with WCOJ-style matching [34]; our GPU execution uses the same style inside each tree-decomposition node. BFS methods expose parallelism but generate large intermediate results, while DFS methods avoid materialization but suffer from workload imbalance, irregular access, and warp divergence. Recent DFS optimizations [12, 20] are specialized for specific patterns and do not support arbitrary query graphs.

GPU Hash Tables. GPU hash tables have been studied for high-throughput key–value workloads. Open-addressing designs (e.g., WarpDrive and WarpCore [18, 19]) resolve collisions by probing alternative buckets. Cuckoo hashing [1, 26] bounds the number of candidate locations per key, leading to bounded-probe lookups, but insertions may trigger eviction chains that increase update overhead. Perfect hashing (e.g., GPH [7]) constructs collision-free mappings to enable one-probe lookups, but is most suitable for static or rarely-updated tables. Our solution maintains intermediate tables with frequent concurrent updates. Therefore, we adopt an open-addressing hash table [19] as our backend implementation.

8 CONCLUSION

In this paper, we presented a high-performance GPU solution for local subgraph counting. We identified the fundamental scalability bottleneck when porting SCOPE to GPU: the tension between massive thread parallelism and limited device memory. To resolve this, we introduced compressed data structures coupled with an insert-failure restart mechanism, allowing our GPU solution to maximize parallelism without risking memory exhaustion. We further optimized throughput via a key mapping strategy for lock-free hash tables. Finally, we generalized these techniques to support arbitrary queries. Extensive experiments on real-world datasets show substantial speedups over the multi-threaded SCOPE implementation.

ACKNOWLEDGMENTS

This work is supported by grants from the Research Grants Council of Hong Kong Special Administrative Region, China (No. PolyU 15205224) and from ByteDance (P0063921).

REFERENCES

[1] Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2009. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH asia*. 1–9.

[2] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. Parallel k-clique counting on gpus. In *ICS*. ACM, New York, NY, USA, 1–14.

[3] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *IPDPS*. IEEE, Los Alamitos, CA, USA, 419–429.

[4] Pablo Barceló, Floris Geerts, Juan Reutter, and Maksimilian Ryschkov. 2021. Graph neural networks with local graph parameters. In *NIPS*. Curran Associates, Inc., Red Hook, NY, USA, 25280–25293.

[5] Diana Cai, Trevor Campbell, and Tamara Broderick. 2016. Edge-exchangeable graphs and sparsity. In *NIPS*, Vol. 29. Curran Associates, Inc., Red Hook, NY, USA, 4249–4257.

[6] Mario Cannataro and Pietro H Guzzi. 2012. *Data management of protein interaction networks*. Vol. 17. John Wiley & Sons, Hoboken, NJ, USA.

[7] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *PACMMOD* 3, 3 (2025), 1–26.

[8] Sudhanshu Chanpuriya, Cameron Musco, Konstantinos Sotiropoulos, and Charalampos Tsourakakis. 2020. Node embeddings and exact low-rank representations of complex networks. In *NIPS*, Vol. 33. Curran Associates, Inc., Red Hook, NY, USA, 13185–13198.

[9] Jingji Chen and Xuehai Qian. 2023. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *ASPLOS*. ACM, 47–61. https://doi.org/10.1145/3567955.3567956

[10] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *OSDI*. USENIX Association, 857–877.

[11] Hanjun Dai, Azade Nazi, Yujia Li, Bo Dai, and Dale Schuurmans. 2020. Scalable deep generative modeling for sparse graphs. In *ICML*. PMLR, Online, 2302–2312.

[12] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. PARSEC: Parallel subgraph enumeration in CUDA. In *IPDPS*. IEEE, Los Alamitos, CA, USA, 168–178.

[13] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *ICDT*. ACM, New York, NY, USA, 8–21.

[14] Kernel Profiling Guide General. 2025. Kernel Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide Accessed: 2026-01-16.

[15] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*. ACM, New York, NY, USA, 1067–1082.

[16] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2022. Exploiting Reuse for GPU Subgraph Enumeration. *TKDE* 34, 9 (2022), 4231–4244.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *NIPS*. Curran Associates, Inc., Red Hook, NY, USA, 22118–22133.

[18] Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2018. WarpDrive: Massively parallel hashing on multi-GPU nodes. In *IPDPS*. IEEE, Los Alamitos, CA, USA, 441–450.

[19] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. Warpcore: A library for fast hash tables on gpus. In *HIPC*. IEEE, Los Alamitos, CA, USA, 11–20.

[20] Samiran Kawtikwar, Mohammad Almasri, Wen-Mei Hwu, Rakesh Nagi, and Jinjun Xiong. 2023. BEEP: Balanced Efficient subgraph Enumeration in Parallel. In *ICPP*. ACM, New York, NY, USA, 142–152.

[21] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned cardinality estimation: An in-depth study. In *SIGMOD*. ACM, New York, NY, USA, 1214–1227.

[22] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook-Shin Han. 2021. Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation. In *SIGMOD*. ACM, New York, NY, USA, 964–976.

[23] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: 2026-05-31.

[24] Itay Levinas, Roy Scherz, and Yoram Louzoun. 2022. BFS-based Distributed Algorithm for Parallel Local-Directed Subgraph Enumeration. *J. Complex Netw.* 10, 6 (2022), cnac051. https://doi.org/10.1093/comnet/cnac051

[25] Qiyan Li and Jeffrey Xu Yu. 2024. Fast Local Subgraph Counting. *PVLDB* 17, 8 (2024), 1967–1980.

[26] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. Dycuckoo: dynamic hash tables on gpus. In *ICDE*. IEEE, Los Alamitos, CA, USA, 744–755.

[27] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2017. Network motif discovery: A GPU approach. *TKDE* 29, 3 (2017), 513–528.

[28] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *SOSP*. ACM, 509–523. https://doi.org/10.1145/3341301.3359633

[29] Ine Melckenbeeck, Pieter Audenaert, Didier Colle, and Mario Pickavet. 2018. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinformatics* 34, 8 (2018), 1372–1380.

[30] Ine Melckenbeeck, Pieter Audenaert, Tom Michoel, Didier Colle, and Mario Pickavet. 2016. An algorithm to automatically generate the combinatorial orbit counting equations. *PloS one* 11, 1 (2016), e0147078.

[31] Ine Melckenbeeck, Pieter Audenaert, Thomas Van Parys, Yves Van De Peer, Didier Colle, and Mario Pickavet. 2019. Optimising orbit counting of arbitrary order by equation selection. *BMC bioinformatics* 20, 1 (2019), 27.

[32] Kernel Profiling Guide Device Memory. 2025. Kernel Profiling Guide: Device Memory. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#device-memory Accessed: 2026-01-16.

[33] Kernel Profiling Guide Source Metrics. 2025. Kernel Profiling Guide: Source Metrics. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#source-metrics Accessed: 2026-01-16.

[34] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[35] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[36] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide*. NVIDIA Corporation. Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide. Accessed: 2026-05-31.

[37] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. 2016. Pte: Enumerating trillion triangles on distributed systems. In *KDD*. ACM, New York, NY, USA, 1115–1124.

[38] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*. ACM, New York, NY, USA, 1099–1114.

[39] Noujan Pashanasangi and C Seshadhri. 2020. Efficiently counting vertex orbits of all 5-vertex subgraphs, by evoke. In *WSDM*. ACM, New York, NY, USA, 447–455.

[40] Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *WWW*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1431–1440.

[41] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.

[42] Chendi Qian, Gaurav Rattan, Floris Geerts, Mathias Niepert, and Christopher Morris. 2022. Ordered subgraph aggregation networks. In *NIPS*, Vol. 35. Curran Associates, Inc., Red Hook, NY, USA, 21030–21045.

[43] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. 2021. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *Comput. Surveys* 54, 2 (2021), 1–36.

[44] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. AAAI Press, Palo Alto, CA, USA, 4292–4293. http://networkrepository.com Accessed: 2026-05-31.

[45] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. *PVLDB* 17, 7 (2024), 1697–1709.

[46] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *PVLDB* 15, 1 (2021), 85–97.

[47] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *SIGMOD*. ACM, New York, NY, USA, 1083–1098.

[48] Xibo Sun and Qiong Luo. 2023. Efficient gpu-accelerated subgraph matching. *PACMMOD* 1, 2 (2023), 1–26.

[49] Charalampos E Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable motif-aware graph clustering. In *WWW*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1451–1460.

[50] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural subgraph counting with wasserstein estimator. In *SIGMOD*. ACM, New York, NY, USA, 160–175.

[51] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed subgraph enumeration with backtracking-based framework. In *ICDE*. IEEE, Los Alamitos, CA, USA, 136–147.

[52] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *SC*. IEEE, Piscataway, NJ, USA, 1–14.

[53] Chang Ye, Yuchen Li, Shixuan Sun, and Wentian Guo. 2024. gsword: Gpu-accelerated sampling for subgraph counting. *PACMMOD* 2, 1 (2024), 1–26.

[54] Hao Yin, Austin R Benson, and Jure Leskovec. 2019. The local closure coefficient: A new perspective on network clustering. In *WSDM*. ACM, New York, NY, USA, 303–311.

[55] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *ICDE*. IEEE, Los Alamitos, CA, USA, 1249–1260.
[56] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. 2020. Distributed subgraph counting: a general approach. *PVLDB* 13, 12 (2020), 2493–2507.
[57] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A learned sketch for subgraph counting. In *SIGMOD*. ACM, New York, NY, USA, 2142–2155.