

Mayura: Exploiting Similarities in Motifs for Temporal Co-Mining

Sanjay Sri Vallabh Singapuram
University of Michigan
Ann Arbor, Michigan, USA
singam@umich.edu

Ronald Dreslinski
University of Michigan
Ann Arbor, Michigan, USA
rdreslin@umich.edu

Nishil Talati
University of Michigan
Ann Arbor, Michigan, USA
talatin@umich.edu

ABSTRACT

Temporal graphs serve as a critical foundation for modeling evolving interactions in domains ranging from financial networks to social media. Mining temporal motifs is essential for applications such as fraud detection, cybersecurity, and dynamic network analysis. However, conventional motif mining approaches treat each query independently, incurring significant redundant computations when similar substructures exist across multiple motifs. In this paper, we propose Mayura, a novel framework that unifies the mining of multiple temporal motifs by exploiting their inherent structural and temporal commonalities. Central to our approach is the *Motif-Group Tree (MG-Tree)*, a hierarchical data structure that organizes related motifs and enables the reuse of common search paths, thereby reducing redundant computation. We propose a co-mining algorithm that leverages the MG-Tree and develop a flexible runtime capable of exploiting both CPU and GPU architectures for scalable performance. Empirical evaluations on diverse real-world datasets demonstrate that Mayura achieves substantial improvements over the state-of-the-art techniques that mine each motif individually, with an average speed-up of 2.4× on the CPU and 1.7× on the GPU, while maintaining the exactness required for high-stakes applications.

PVLDB Reference Format:

Sanjay Sri Vallabh Singapuram, Ronald Dreslinski, and Nishil Talati.
Mayura: Exploiting Similarities in Motifs for Temporal Co-Mining. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Temporal graphs have become a fundamental abstraction for modeling dynamic interactions in domains ranging from financial transaction networks to social media ecosystems [5, 12, 43, 44, 49, 58]. These graphs capture not only topological relationships but also the temporal evolution of interactions, enabling the analysis of complex phenomena such as information diffusion, fraud patterns, and network dynamics. With the advent of large-scale temporal datasets—exceeding billions of edges in domains like blockchain transactions and communication networks—the need for efficient analytical primitives has never been more critical [17]. Traditional graph mining techniques, designed for static graphs, fail to account for the temporal ordering and time-window constraints inherent in

real-world systems, necessitating specialized approaches for temporal pattern discovery [11, 18, 19, 31, 32, 42, 45, 48, 56].

A cornerstone of analyzing relationships in temporal graph is temporal motif mining, which identifies and enumerates sequences of time-constrained edges whose structure is governed by a motif (e.g., 3-cycles) [34]. These motifs can represent meaningful relationships in the underlying temporal graph, such as a pattern of suspicious transactions between bank accounts within a short period of time, thus helping with fraud detection in financial networks [15]. Applications of temporal mining also include cybersecurity threat analysis [14, 30], social behavior modeling [5, 58], and monitoring energy disaggregation in electrical grids [41]. Existing systems focus on mining individual motifs, while real-world workloads often need to process queries with multiple motifs that overlap structurally and temporally. For instance, anti-money laundering investigations [3, 6, 12, 44] often require simultaneous detection of multiple transaction patterns across shared subsets of edges. Current approaches thus incur redundant computations as they need to repeatedly traverse the graph for each motif.

While multi-query optimization (MQO) techniques for static graphs and approximate temporal mining offer partial solutions, they prove inadequate because the techniques used to exploit similarities are not applicable to temporal motif mining [20, 35]. Approximate counting [27, 33, 36, 38, 55] sacrifices accuracy for performance, which cannot be used in high-stakes domains like finance, where the exact identification of crime is necessary.

This paper addresses aforementioned limitations by proposing **Mayura**, the first system to enable efficient co-mining of temporal motifs. Mayura accomplishes this by using a data-structure called the *Motif-Group Tree (MG-Tree)*, a hierarchical representation of motifs that captures edge-level commonalities, enabling shared search path exploration and eliminating redundant computations. The temporal mining algorithm is then adapted to search for matches guided by the MG-Tree instead of a single motif. Mayura supports efficient co-mining on both CPUs and GPUs, balancing the workload across multiple threads (and warps) and exploiting the hierarchical parallelism exposed by the MG-Tree. Mayura also generates code that optimizes execution for a specific MG-tree to improve instruction throughput and reduce the architectural resource footprint.

Co-mining presents several critical system-level challenges that must be addressed to maximize the efficacy of this approach. Load balancing emerges as a fundamental challenge due to the inherent irregularity of graph workloads, where different search paths exhibit vastly different computational requirements, necessitating sophisticated work distribution strategies. Co-mining on the GPU presents additional challenges. The irregularity of the graph workload makes the GPU implementation susceptible to control-flow divergence, leading to serialized execution and lower performance, requiring the control-flow to be more streamlined than the baseline.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

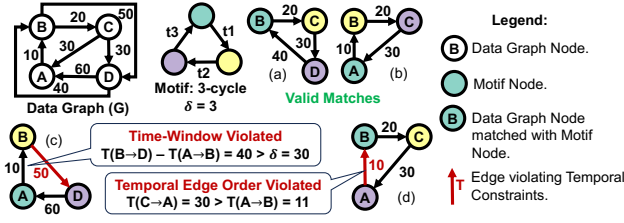


Figure 1: Example of mining a 3-cycle motif within a temporal Data Graph (G). Data vertices (e.g. ⑤) in the matches are color-coded (⑤) to their corresponding motif vertex (●).

The GPU also poses a resource bottleneck with limited register capacity per thread when maintaining context for multiple motifs simultaneously, leading to reduced occupancy and suboptimal hardware utilization, and must be addressed by careful selection of instructions to minimize register usage.

Our comprehensive evaluation demonstrates Mayura’s significant performance improvements across diverse temporal datasets. On a 40-core Intel Xeon CPU, Mayura achieves average speedups between 1.8-3.7× over multiple datasets, with peak acceleration of 8.8× over a bipartite graph. On an NVIDIA A40 GPU, Mayura achieves average speedups between 1.3-3.1× over many datasets, and 7.6× maximum speedup despite architectural constraints. These gains stem from Mayura’s core contributions: dynamic instruction counts reduce by 1.6-4.5× through the MG-Tree-guided search space pruning and code optimization, while motif-specific code generation alleviates 87-94% of warp divergence penalties on the GPU. These results validate that strategic co-mining of temporal motifs enables significant efficiency gains while preserving exactness for critical applications. Mayura makes the following contributions.

- (1) **The MG-Tree:** A hierarchical data-structure that captures structural and temporal similarities across motifs, helping unearth shared search paths.
- (2) **Temporal Co-Mining Algorithm:** The first exact algorithm capable of simultaneously mining multiple motifs.
- (3) **Multi-Backend Runtime:** A unified execution framework supporting both CPU and GPU backends.
- (4) **GPU-Specific Code Optimizations:** Predicated execution, register-bound context mapping and expression simplification to reduce warp divergence and improve occupancy.
- (5) **Hierarchical Parallelization:** Two complementary GPU optimizations—*sibling-splitting* and *multi-offload* that expose parallelism across the MG-tree hierarchy.

2 BACKGROUND

2.1 Problem definition

A **Temporal Graph** is defined as an ordered collection of *temporal edges*, where each *temporal edge* is a directed connection between two vertices with an associated timestamp. Formally, a temporal graph $G = (V_G, E_G)$ is a set of vertices V_G connected by a list of n temporal edges $E_G = (u_i, v_i, t_i)_{i=1}^n | u_i, v_i \in V_G$, where u_i and v_i are the source and destination vertices of the edge (u_i, v_i) , respectively, and $t_i \in \mathbb{R}^+$ is the edge’s timestamp. The edges are

chronologically ordered with unique timestamps. Additionally, both vertices and edges have optional discrete or continuous attributes (e.g., node/edge types).

A δ -**Temporal Motif** $M = (V_M, E_M)$ is defined as an ordered sequence of m edges, where $E_M = (u_i, v_i, t_i)_{i=1}^m | u_i, v_i \in V_M$, that occur within a specified time window of length $\delta \in \mathbb{R}^+$ [30]. The edges of G that match with M must be temporally ordered ($t_1 < t_2 \dots < t_m$) and the entire sequence must occur within a time-window δ , i.e. $(t_m - t_1 \leq \delta)$. The label of a motif-edge in all figures indicates the edge’s relative order.

Temporal Motif Mining is the task of mining instances of a δ -temporal motif within a temporal graph. This process can yield two types of results: either a comprehensive list of all matching motifs (enumeration) or a tally of their occurrences (counting). Our formulation of this problem requires finding one-to-one correspondences between the vertices of motif and subgraphs within the temporal graph being mined, which is also known as isomorphism-based mining [57]. While some related work in the field of temporal mining adopt slightly different notions of a match, e.g., mapping sets of edges of the temporal graph to the edges of the motif [24], sequences of time-stamped events across snapshots of static networks [47] or homomorphism between edges of the motif and a match [8], we restrict the scope of this work to a strict isomorphism-based structural definition of a match based on prior works [30, 34, 57]. We therefore use “mining” to denote identifying instances of a pattern, as opposed to very early literature that equated “mining” with discovering novel patterns [10].

The problem of mining δ temporal motifs is similar to SQL query execution in traditional databases, where the temporal graph can be thought of as a database and the motif and time-window are analogous to the query. Temporal mining can then be expressed as nested JOIN operations to match edges in the data graph to edges in the motif by using a vertex as the common key, and WHERE clauses to filter out edges that violate temporal constraints.

The Walk-Through Example in Fig. 1 illustrates the different aspects of temporal motif mining introduced above. The temporal data graph G has four vertices, A through D, and seven edges marked with timestamps using the same unit, e.g., seconds. The motif’s vertices are color-coded to map with their corresponding data graph vertex in a match, with the timestamp of edges indicating their relative temporal order: $T(\text{●} \rightarrow \text{●}) < T(\text{●} \rightarrow \text{●}) < T(\text{●} \rightarrow \text{●})$. Given the temporal graph G , a 3-cycle motif to be mined and a time-window $\delta = 30$, Fig. 1(a)-(d) illustrates different sub-graphs that are valid and invalid matches, along with the rationale for their classification. The difference in the timestamp between the first and the last edges is 2 for 1(a) and 1(b), making them valid matches ($2 < \delta = 30$). Fig. 1(c) is an invalid match because its second edge occurs more than 30 time-steps after the first edge, placing it outside the permissible time-window. 1(c) violates the time-window since the second and third edges occur more than 30 time-steps after the first edge. Fig. 1(d) violates the temporal ordering constraint: $T(\text{●} \rightarrow \text{●}) = 30 > T(\text{●} \rightarrow \text{●}) = 10$.

```

1: Inputs : Temporal ta Graph G, Motif M, and Time-Window size  $\delta$ 
2: Output Variables : count  $\leftarrow 0$ ; matches  $\leftarrow \emptyset$ ;
3: Book-Keeping Context : e_stack[ $\cdot$ ]  $\leftarrow \emptyset$ ; m2g[ $\cdot$ ]  $\leftarrow -1$ ; g2m[ $\cdot$ ]  $\leftarrow -1$ ; incnt[ $\cdot$ ]  $\leftarrow 0$ ;
4: procedure TEMPORALMINING(TemporalGraph G, Motif M, Int  $\delta$ )
5:   MATCHEDGE(G, M,  $\delta$ , 0); return count, matches;  $\triangleright$  Start by matching first motif edge.
6: function MATCHEDGE(TemporalGraph G, Motif M, Int  $\delta$ , Int eM)
7:   if eM = |E(M)| then  $\triangleright$  Match Found when eM reaches number of motif edges.
8:     count  $\leftarrow$  count + 1; matches.append(e_stack); return
9:   uM, vM  $\leftarrow$  M.edges[eM]; uG  $\leftarrow$  m2g[uM]; vG  $\leftarrow$  m2g[vM]  $\triangleright$  Get uG  $\leftrightarrow$  uM map.
10:  cands  $\leftarrow$  (uG  $\neq -1$ ) ? N(uG) : G.edges  $\triangleright$  All edges are candidates for unmapped uG.
11:  for edge G  $\in$  cands do
12:    if eM > 0 and (edgeG.t < e_stack[eM - 1].t or edgeG.t - e_stack[0].t >  $\delta$ ) then
13:      continue  $\triangleright$  Ensure Time-Edge-Order and Time-Window Constraints.
14:    if vG  $\neq -1$  and edgeG.v  $\neq$  vG then continue  $\triangleright$  Ensure Structural Constraints.
15:    e_stack.append(edgeG);
16:    ROLLONEDGE(uM, vM, edgeG.u, edgeG.v)  $\triangleright$  Book-keep (uM, vM) to edgeG map.
17:    MATCHEDGE(G, M,  $\delta$ , eM + 1)  $\triangleright$  Expand Search-Tree to match next M edge recursively.
18:    e_stack.pop(); ROLLBACKEDGE(edgeG.u, edgeG.v);  $\triangleright$  Remove edgeG's mapping.
19: function ROLLONEDGE(Int uM, Int vM, Int uG, Int vG)
20:   m2g[uM]  $\leftarrow$  uG; g2m[uG]  $\leftarrow$  uM; m2g[vM]  $\leftarrow$  vG; g2m[vG]  $\leftarrow$  vM
21:   incnt[uG]  $\leftarrow$  incnt[uG] + 1; incnt[vG]  $\leftarrow$  incnt[vG] + 1
22: function ROLLBACKEDGE(Int uG, Int vG)
23:   uG, vG  $\leftarrow$  edge; incnt[uG]  $\leftarrow$  incnt[uG] - 1 incnt[vG]  $\leftarrow$  incnt[vG] - 1
24:   if incnt[uG] = 0 then uM  $\leftarrow$  g2m[uG]; g2m[uG] = m2g[uM] = -1
25:   if incnt[vG] = 0 then vM  $\leftarrow$  g2m[vG]; g2m[vG] = m2g[vM] = -1

```

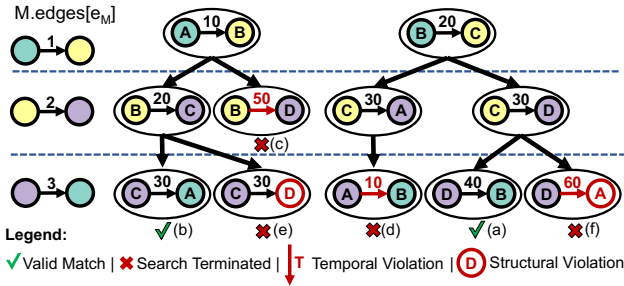


Figure 2: Search-Tree to mine a 3-cycle in Data Graph G in Fig. 1. Labeled leaf nodes (a-d) are search-paths of corresponding valid/invalid matches in Fig. 1.

2.2 Algorithmic Prior Work

Temporal motif mining approaches fall into two categories: exact algorithms that enumerate all matches through subgraph isomorphism with temporal constraints [30, 34, 57], and approximate methods that estimate counts via sampling or sketching [27, 33, 36, 38, 55]. While approximate techniques scale better for large graphs, exact methods remain crucial for applications requiring complete enumeration, such as fraudulent activity identification in financial networks [15] or insider threat identification [14, 30]. Notably, many approximate methods still leverage exact algorithms as subroutines for local pattern matching [26, 38]. Given these considerations, *our work focuses on exact algorithms*, with a brief discussion of approximate techniques deferred to § 8.

Paranjape *et al.* [34] formalized δ -temporal motif mining and proposed a two-phase algorithm: 1) enumerating static isomorphic subgraphs, then 2) verifying temporal constraints. The first step uses existing static subgraph / pattern mining methods that identify matches that are structurally equivalent to the motif, ignoring

temporal constraints. This potentially leads to unnecessary computational overhead, as structurally compliant candidates may not satisfy temporal requirements [57]. Subsequent work by Mackey *et al.* [30] improved efficiency by pruning temporally invalid candidates before expanding the entire subgraph. Everest [57] adapted this approach for GPUs with a state-of-art warp-level parallelization of candidate exploration. The essence of Mackey’s algorithm [30] is captured in Algorithm 1, with a subsequent discussion on Everest’s [57] distinctions from this approach.

Data-Structures: The algorithm employs a set of book-keeping variables (line 3) to map a motif edge and a graph edge. Specifically, "m2g" and "g2m" facilitate bidirectional mapping between motif and graph vertices, "e_stack" maintains a temporally ordered stack of all currently matched edges, while "incnt" tracks the number of matched edges incident on each graph vertex.

Search-Tree: Mining the graph can be conceptualized as a tree, where nodes within a level represent candidate edges for the corresponding motif edge, and the parent corresponds to the graph edge matched to the preceding motif edge. Fig. 2 illustrates a few search-trees for mining a 3-cycle in the data graph G (Fig. 1), with individual paths labeled to correspond to matches Fig. 1(a-d).

Algorithm 1 begins by mapping the first motif edge (line 5, $e_M = 0$), considering all edges in the data graph as potential candidates due to the absence of pre-existing vertex mappings. It then employs a depth-first exploration strategy to recursively expand the search tree for subsequent motif edges (line 17, $e_M > 0$). When exploring a new level, the algorithm prunes the candidate list to the out-edges of a potentially mapped source vertex u_G . The candidates are further pruned based on temporal and structural constraints (line 13-14). Fig. 2(c,d) illustrate the search tree being pruned due to temporal violations, corresponding to the invalid matches Fig. 1(c,d). The structural constraint enforces a bijective mapping between the destination vertex, v_M , and graph vertex, v_G . Search paths terminating at Fig. 2(e,f) illustrate structural violations since the last edge must finish the cycle by ending at the first vertex (A for (e) and B for (f)). After passing the constraint checks, the book-keeping context is updated to capture the mapping between e_M and $edge_G$ (line 16), and recursively proceeds to the next motif edge. Once all motif edges have been matched (Fig. 2(a,b)), the algorithm records the match by either incrementing a counter or adding it to an enumeration list (line 8). This algorithm can be parallelized over candidates for the first edge, but becomes challenging for subsequent edges due to the sequential nature of candidate generation (line 11). Everest [57] extends Algo. 1 to GPUs by storing the candidate list as a range of edges. By distributing these ranges among threads within a GPU warp, Everest [57] enables a massively parallel exploration of candidates across all levels of the search-tree.

3 A CASE FOR MULTI-QUERY EXECUTION

This section motivates the need for efficient multi-query execution and evaluates the effectiveness of existing techniques in addressing the challenges of temporal motif mining.

Modern analytical workloads increasingly require concurrent execution of multiple queries across domains ranging from financial

fraud detection to social network analysis. Anti-money laundering systems must simultaneously track diverse transaction patterns [3, 6, 12, 44], while social platforms analyze user engagement through parallel interaction queries [5, 58]. Traditional single-query optimization proves inadequate for these workloads due to *redundant computations across overlapping queries*.

3.1 Collective Queries in Traditional Databases

The challenge of multi-query optimization (MQO) has been studied since Sellis’ seminal work on identifying common subexpressions [40], though optimal planning remains NP-hard [39]. Contemporary approaches fall into two categories: 1) physical optimizations that make the underlying system more efficient (*e.g.*, shared data access patterns via scan-sharing [4, 9] or resource-aware scheduling [2, 7]), and 2) algebraic transformations that algorithmically reduce the amount of work [50]. Ren and Wang [35] pioneered MQO for isomorphic-pattern mining, by reducing redundant computations by identifying shared structures across queries. Subgraph Morphing [20], a hybrid approach combining algebraic and physical optimizations, decomposes query patterns into alternative patterns that are less computationally expensive to mine.

3.2 Opportunity: Commonality in Temporal Motif Queries

Building upon insights gleaned from prior work, we explore potential ways to identify and reduce redundant computation. The techniques proposed previously [20, 35] could be adapted to exploit structural similarities among temporal queries, but not without challenges. Ren and Wang [35] can be used to first identify isomorphic-matches with query motifs and then filtered to comply with temporal constraints, leading to unnecessary exploration of a large search-space [34]. Subgraph Morphing [20] exploits symmetry in graph isomorphism to reduce redundant searches, may produce matches that violate temporal constraints in the context of temporal motif mining (*e.g.*, Fig. 1(b) and (d)). This issue is exacerbated when two vertices have multiple edges between them, leading to a combinatorial explosion in the number of potential matches requiring a large memory footprint for enumeration.

Given the challenges associated with efficiently adapting existing techniques for temporal motif mining, we allude to an approach specifically tailored to this domain. Consider Fig. 3, which illustrates the 3-cycle, 4-cycle, and M4 motifs (Fig. 1). Observe that the motifs share the same first two edges, $t_1 \rightarrow t_2$ and $t_2 \rightarrow t_3$, indicating that the ideal mining algorithm must visit these edges in the search-tree for either a 3-cycle or a 4-cycle before proceeding to expand to the third edge. This observation suggests that we can eliminate redundant computation along the common path. By prioritizing edge traversal in a chronological order, we establish a natural heuristic that favors exploring common paths before diverging into specific motif searches. In subsequent section, we generalize this concept to accommodate more temporally and structurally complex common paths, laying the foundation for our efficient co-mining approach.

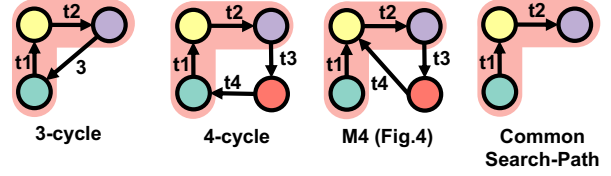


Figure 3: Common search-path between 3-cycle, 4-cycle and M4 motifs.

4 MAYURA DESIGN

This section presents the design of Mayura, outlining its core objectives, workflow, key algorithmic components and the runtime for CPU/GPU backend. We introduce the Motif-Group Tree (MG-Tree) generation algorithm and the co-mining algorithm, which form the foundation of our approach.

4.1 Design Goals

Mayura is designed with three primary objectives to address the challenges in multi-query execution for temporal motif mining.

Efficient Motif Co-Mining. While existing approaches to temporal motif mining have focused on optimizing mining a single motif, our observation in §3.2 presents an opportunity to identify redundant computations when mining multiple motifs. We aim to minimize these redundant computations by exploiting structural and temporal similarities among motifs within the query set, reducing the overall workload associated with mining multiple motifs.

Multi-Backend Support. To ensure broad accessibility, one of our goals is to support seamless operation on both GPU and CPU platforms. Recognizing the memory constraints associated with GPUs and that not all users have access to high-performance GPUs, our design allows users the flexibility to execute motif mining tasks on CPU resources when necessary.

High Performance Optimizations. While co-mining presents clear opportunities for performance improvements, its implementation introduces specific challenges that must be addressed. For instance, the additional context and control-flow introduced to enable co-mining on GPU threads can potentially reduce performance due to the additional register footprint and warp-divergence from threads in the same warp mining different motifs. Our goal is to address these challenges (§5), ensuring that the system not only capitalizes on the benefits of co-mining but also maintains high performance throughout the mining process.

4.2 Mayura Workflow

Our system takes as input a user query (Fig. 4) that specifies the temporal data graph to mine (Fig. 5 ①), a group of motifs (*i.e.*, motif group), time-window δ , backend choice (CPU or GPU), and whether matches need to be counted or enumerated. The output is either the per-motif count or enumeration. The workflow of Mayura is structured into three distinct phases: Compile-Time, Data-Loading and Runtime, visualized by Fig.5. The Compile-Time phase generates and compiles the code that implements the co-mining algorithm, while the Data-Loading phase operates concurrently to load the dataset into (CPU/GPU) memory. The Runtime phase executes the compiled code to mine the loaded dataset.

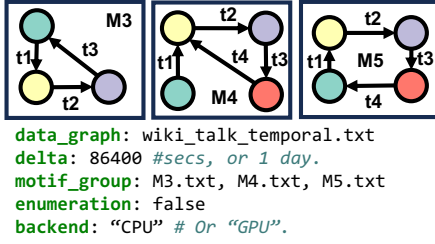


Figure 4: Example User Query.

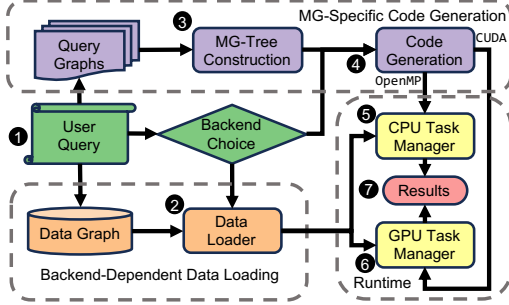


Figure 5: Overview of Mayura's workflow.

Data-Loading and Compile-Time Phases consists of mechanisms for 1) Data Preprocessing (2), 2) Motif-Group Tree (MG-Tree) Construction (3), and 3) Backend-Specific Code Generation and Compilation (4). The temporal data graph specified by the query (1) is preprocessed from an edge-list format (u_i, v_i, t_i) to an adjacency-list format stored in a CSR-like structure, with edges sorted in ascending order of timestamps (2). The system constructs a hierarchical representation of the query motifs in the motif group, *i.e.*, the MG-Tree, capturing structural and temporal similarities (3). The system then generates optimized code tailored to the specific MG-Tree and backend, utilizing C++ with OpenMP for CPU execution or CUDA C for GPU execution (4).

Runtime Phase dispatches the compiled code and preprocessed data to the appropriate backend's task manager (5 for CPU, 6 for GPU). To address the inherent load imbalance in graph workloads, both CPU and GPU task managers implement sophisticated load balancing strategies (§4.5). Upon completion, the system aggregates the results, providing either motif counts or enumerations as specified in the query (7).

4.3 Motif-Group Tree Construction

We propose a hierarchical data structure called Motif-Group Tree (MG-Tree) that captures structural and temporal similarities among query motifs to facilitate efficient co-mining. By grouping motifs that share overlapping edges (with the same relative order) in a hierarchical structure, the MG-Tree enables the co-mining algorithm to reuse computations along the shared search paths and eliminating redundant work. This hierarchical structure also exposes algorithmic parallelism to improve concurrency. While the MG-Tree is not the first to exploit structural similarities to accelerate matching [21, 52], we are the first to propose such a solution

for temporal mining. We define the MG-Tree by defining its composition and the constraints that define the relationship between parent and children nodes,

MG-TREE DEFINITION: For a group of temporal motifs $MG = \{M_1, M_2, \dots, M_k\}$, the MG-Tree MGT is defined as a tree of Nodes that capture the similarities among motifs in MG , rooted at N_{root} .

Node Composition: Any Node $N \in MGT$ is composed of the following 3 members: C_N , $Children(N)$ and Q_N ,

C_N : A motif with edges common across all descendants $C_{N_{desc}}$, *i.e.* a prefix for $C_{N_{desc}}$ with their first $|C_N|$ edges being equal to C_N .

$N_{child} \in Children(N)$: Immediate descendants constructed by extending C_N , where C_N is the longest non-trivial prefix of $C_{N_{child}}$.

Q_N : The reference to a query motif $M_i \in MG$ when C_N is equivalent to M_i , else is \emptyset . $\forall M_i \in MG, \exists N \in MGT \mid Q_N = M_i$

Root Node: $N_{root} \in MGT$ whose common motif $C_{N_{root}}$ has edges common across all motifs in MG .

The MG-Tree construction algorithm (Algo. 2) begins by invoking the `CONSTRUCTMGTREE` procedure on the motif group MG . For the sake of brevity and simplicity, we refer to the temporal order of a motif edge as its timestamp. A TMap is generated for each motif, mapping timestamps to the corresponding edges. Nodes with children are known as Intermediate Nodes (Intr.), and those without are Leaf Nodes. Members of the motif group are populated as leaf Nodes in the MG-Tree, with their C_N and Q_N references set to the member (line 9). Upon reaching a leaf Node N during co-mining, the search is limited to mine only Q_N . The algorithm then proceeds in a recursive manner to build the MG-Tree, starting from the first edge in all motifs (line 7). The motifs are grouped together based on the source and destination of the edges at timestamp T (line 21). Motifs in singleton groups are inserted as a list of children into their parent Node (p_gid). Undivided groups, *i.e.*, all motifs in the input group have the same edge at $T - 1$ and T , end up reusing the node created at $T - 1$ (or potentially before) (line 31). A new Intr. Node is created for all other groups, representing motifs encountered during the search process but not necessarily counted or enumerated like Q_N . The algorithm also eliminates redundant work when C_N is identical to a Q_N in the `child_group`. The new Intr. Node is then added as a child to its parent (line 38). The MG-Tree for each child group is recursively constructed and attached to the final MG-Tree (line 36), with the recursion terminating at Leaf Nodes.

Walkthrough Example: Consider the motif group $[M3, M4, M5]$ from Fig. 4 as an input to the algorithm. Fig. 6 visualizes the construction of the MG-Tree in Fig. 7. After setting up TMap and the MG-Tree with leaf Nodes, `CREATETREE` is invoked on the motif group. Since all motifs have the same edge at $T = 1$, *i.e.*, $\text{blue} \rightarrow \text{yellow}$, the `child_group` reuses N_{root} at `c_gid = 0`, referred to as Intermediate Node $I1$. N_{root} 's C_N is set to contain the single edge, $\text{blue} \rightarrow \text{yellow}$, and the Q_N is left empty since none of motifs resemble C_N . With the recursive call to `CREATETREE`, all the motifs end up being grouped together at $T = 2$ as well since they still share an edge, $\text{yellow} \rightarrow \text{purple}$. The C_N for $I1$ is reset to $\text{blue} \rightarrow \text{yellow} \rightarrow \text{purple}$. At $T = 3$, $M3$ is grouped separately from $M4$ and $M5$ since its edge, $\text{purple} \rightarrow \text{blue}$, is different from that of $M4$ and $M5$, $\text{purple} \rightarrow \text{red}$. $M3$ is added as a child for $I1$. A new Intr. Node, $I2$, is created for $M4$ and $M5$ with the C_N , $\text{blue} \rightarrow \text{yellow} \rightarrow \text{purple} \rightarrow \text{red}$. With $I2$ as a parent Node, `CREATETREE` is

called on [M4,M5] and ends up separating them since they have different edges at $T = 4$, and are added as children to I2. Observe that if we had to build an MG-tree only for [M4,M5], it would have been the tree rooted at I2. This hierarchical representation enables the exploitation of similarities between motifs at various granularities, reducing the overall computational workload.

Algorithm 2 MG-Tree Construction

```

1: Initialize Context
2: TMap  $\leftarrow \emptyset$ ; mg_tree  $\leftarrow \emptyset$ ; unique_gid  $\leftarrow 0$ 
3: Input: List of unique motifs; Output: MG-Tree MGT
4: procedure CONSTRUCTMGTREE(List[Motif] MG)
5:   for  $M \in MG$  do
6:     TMap[M]  $\leftarrow$  GRAPHToTMAP(M); INSERTMOTIF(M);
7:   root_gid  $\leftarrow$  GETNEWUNIQUEGID()
8:   N_root  $\leftarrow$  MGT[root_gid]; Children(N_root).clear(); C_N_root  $\leftarrow$  Q_N_root  $\leftarrow \emptyset$ 
9:   CREATETREE(1, GETNEWUNIQUEGID(), MG)
10:  return mg_tree

11: function INSERTMOTIF(Motif M)
12:  N  $\leftarrow$  MGT[M.name]; C_N  $\leftarrow$  Q_N  $\leftarrow$  M; Children(N)  $\leftarrow \emptyset$ 

13: function GRAPHToTMAP(Motif M)
14:  tmap  $\leftarrow \emptyset$ ;
15:  for  $e \in E(M)$  do tmap[e.t]  $\leftarrow (e.u, e.v)$ 
16:  return tmap

17: function GETNEWUNIQUEGID()
18:  new_gid  $\leftarrow$  unique_gid; unique_gid  $\leftarrow$  unique_gid + 1; return STR(new_gid)

19: function CREATETREE(Time T, Str p_gid, List[Motif] motif_group)
20:  edge_group  $\leftarrow \emptyset$ ; Nparent  $\leftarrow$  MGT[p_gid]
21:  for  $M \in$  motif_group do
22:    if |TMap[M]| < T then pass
23:    edges  $\leftarrow$  TMap[M]; e  $\leftarrow$  edges[T]; edge_group[e]  $\leftarrow$  edge_group[e]  $\cup \{M\}$ 
24:  for (e, child_group)  $\in$  edge_group do
25:    q_N  $\leftarrow \emptyset$ 
26:    for  $M \in$  child_group do
27:      if |E(M)| = T then
28:        Q_Nparent  $\leftarrow$  M; break
29:    if |child_group| = 1 then
30:      INSERTCHILD(p_gid, child_group[0])
31:    else
32:      if motif_group = child_group then
33:        c_gid  $\leftarrow$  p_gid
34:      else
35:        c_gid  $\leftarrow$  GETNEWUNIQUEGID()
36:      common_edges  $\leftarrow$  TMap[motif_group[RANDOM(1, |motif_group|)]] [1:T+1]
37:      N  $\leftarrow$  MGT[c_gid]; C_N  $\leftarrow$  MOTIF(common_edges); Q_N  $\leftarrow \emptyset$ 
38:      INSERTCHILD(p_gid, c_gid); CREATETREE(T+1, c_gid, child_group)

39: function INSERTCHILD(Str p_gid, Str child_id)
40:  if p_gid  $\neq$  child_id then
41:    mg_tree[p_gid].children  $\leftarrow$  mg_tree[p_gid].children  $\cup \{child_id\}$ 

```

4.4 Mayura Co-Mining Algorithm

The co-mining algorithm extends the temporal motif mining algorithm (Algo. 1) by using the MG-Tree, instead of a single motif, to guide the expansion of the search tree. The pseudo-code of the algorithm is outlined in Algo 3, focusing only on its modifications from Algo. 1. The core mechanism for mining a single motif-edge remains unchanged (lines 12- 14). The algorithm begins by mining $C_{N_{root}}$. Upon detecting a match for C_N , it is counted if the Node also represents a query motif (line. 6). Subsequently, the algorithm moves onto mining the children of N by using C_N 's match as a partial match (line.13). By expressing co-mining in this recursive manner, we are able to reuse most of the mechanisms used in Algo 1. **Walkthrough Example:** Fig. 8 illustrates a small portion of the search-tree explored by the Algo. 3 when mining the motif group [M3,M4,M5] on the Data Graph G in Fig. 1. The background colors

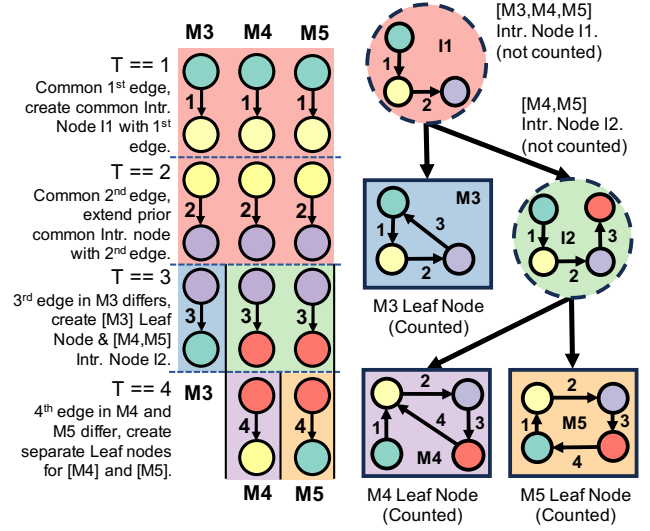


Figure 6: Visualizing MG-Tree Construction.

Figure 7: MG-Tree of motifs in Fig. 4.

Algorithm 3 Co-Mining Algorithm

```

Inputs: Data Graph G, root of MG-Tree and time-window  $\delta$ 
Outputs: Counts and matches of motifs in MG-Tree.

1: procedure TEMPORALCoMINING(TemporalGraph G, Node N_root, Int  $\delta$ )
2:  CoMATCHEDGE(G, N_root,  $\delta$ , 0); return count, matches;
3: function CoMATCHEDGE(TemporalGraph G, Node N, Int  $\delta$ , Int  $e_M$ )
4:  M  $\leftarrow$  C_N
5:  if  $e_M = |E(M)|$  then
6:    if Q_N  $\neq \emptyset$  then
7:      count[Q_N]  $\leftarrow$  count[Q_N] + 1; matches[Q_N].append(e_stack);
8:    for C_N_child  $\in$  Children(N) do
9:      CoMATCHEDGE(G, C_N_child,  $\delta$ , e_M)
10:  return
11:  ... > Unmodified MATCHEDGE pseudocode from lines 9 to 16 in Algorithm 1
12:  CoMATCHEDGE(G, node,  $\delta$ , e_M + 1)
13:  ... > Remaining MATCHEDGE and other pseudocode from line 18 in Algorithm 1

```

correspond to that of the corresponding motif in the MG-tree (Fig. 7) being explored by the search-tree. The motif edges on the sides of the tree at each level indicate the motif edge being mined at that particular level. Since the search tree is for an MG-Tree, different motif edges can be mined at the same level for different sibling Nodes. The search tree starts with identifying matches for Node I1 (top two levels). With matches of I1 as partial matches, the search-tree expands into mining the M3 leaf Node and I2 Intr. Node, eventually terminating at the leaf Nodes: M3,M4 or M5.

4.5 Mayura Runtime

The system initiates the runtime phase by enabling the backend specified by the user. This backend receives the MG-Tree, δ , and pointers to the temporal data graph located within its memory. Both CPU and GPU backends parallelize the co-mining algorithm across multiple threads, employing task managers to handle work-scheduling and load-balancing. While global inputs such as the MG-Tree and δ are shared, the search-tree context (e.g., vertex maps,

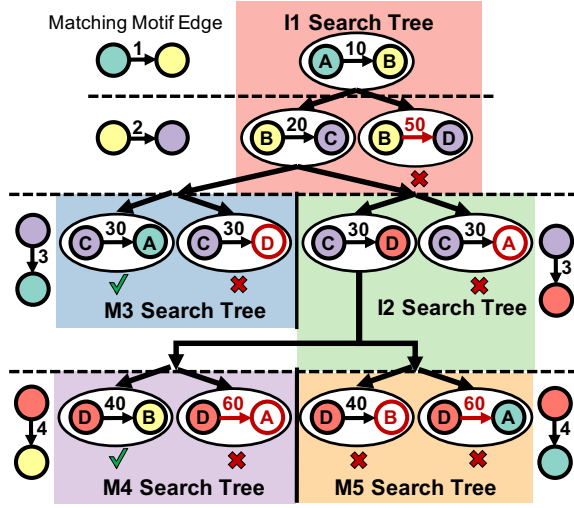


Figure 8: Search-Tree to mine motifs M3,M4 and M5 reflects the hierarchical structure of their MG-Tree (Fig. 6).

counters) is maintained locally within each thread to eliminate contention and maximize parallelism.

The CPU task-manager distributes candidate edges for the first edge across all threads, and utilizes dynamic task scheduling for load balancing. The GPU task-manager employs a two-level policy, which distributes these candidate edges across warps and then splits the search-tree within a warp [57]. This policy is also used to load-balance across threads within a warp (intra-warp) and the across warps (inter-warp). Such a multi-granular policy is necessary given the higher thread count and lower computational capability of individual GPU threads compared to CPU threads. Intra-warp balancing involves periodic polling of idle threads and candidate redistribution using warp-level primitives. Similarly, inter-warp load balancing redistributes the candidates by polling for idle warps across the GPU. Active threads’ search contexts are dumped and then redistributed across all warps on the GPU before resuming mining. Fig. 9 visualizes the runtime-context held within a thread, the CPU Runtime’s load-balancing across CPU threads, and the GPU runtime with its two-tiered balancing 1) within a warp at epoch #e and #e+1, and 2) across warps between two epochs.

Visualizing Intra-Warp Load-balancing: Fig. 10 illustrates the intra-warp load balancing mechanism during co-mining the MG-Tree in Fig. 7, displaying the search-tree and candidates for the second edge in I1 and third edge in M3 or I2. Initially, a warp with four threads has only one active thread (Thread 0) processing candidates (E5-E7) for leaf node M3 (Stage 1.a). The load balancer redistributes these candidates across threads 0-3 to maximize parallelism (Stage 1.b). At a later point, when thread 0 runs out of candidates for M3, it transitions to mining sibling Node I2 and generates a new candidate list (E12...E15). Unlike thread 0, threads 1 and 2 are restricted from mining any sibling of M3 (i.e., I2) in order to prevent double-counting of matches, since thread 0 has already started working on candidates for I2. Subsequent load balancing (Stage 2.a) redistributes I2 candidates from Thread 0 across all four threads, achieving full warp utilization (Stage 2.b)

Visualizing Inter-Warp Load Balancing: Fig. 11 demonstrates the inter-warp load-balancing strategy, the second part of the two-level load-balancing strategy, when mining the MG-Tree in Fig. 7. Upon detecting a threshold number of idle warps, the system interrupts the mining process to make all active threads save their context (i.e., search-tree, book-keeping variables etc.) and exit. In Fig. 10, an active thread is saving its context to the fifth position in the context array located in global memory. This context indicates that the thread is currently exploring candidates for the last edge in M3 (E10,E11,E12), with (E1,E2,E3) and (E4-E7) as candidates for the first and second edge of I2 respectively. Inter-warp load-balancing punctuates periods/epochs of continuous mining with gaps to redistribute the workload. While the search-tree is duplicated across warps with the same source context, the candidates are partitioned across warps. It is possible that different threads in the same warp are assigned to work on search-trees distributed from different contexts. To prevent double counting, the edges that form the search tree are exclusively explored on one thread, with these considered as only a part of the search tree in other threads. For instance, the threads in warp #1 and #2 immediately skip E10 to process E11 and E12 respectively. In addition to skipping E10, the thread in warp #3 also skips E4 as a candidate for the second edge in I2 and moves to considering E6.

Implementing Load-Balancing: Mayura periodically monitors the load-balance rather than continuously calculating the balance factor, owing to the high latency of thread-level synchronization operations. Intra-warp load balancing (Fig. 10) monitors thread status every (say) INTRA_INTRVL iterations, with threads voting to redistribute work when idle threads coexist with active ones. Inter-warp load balancing (Fig. 11) uses global memory to track warp idleness across the GPU every (say) INTER_INTRVL iterations, triggering redistribution when a threshold of idle warps is detected. Since global memory access is more expensive than warp-level synchronization, inter-warp monitoring occurs at longer intervals ($INTER_INTRVL > INTRA_INTRVL$), creating the two-tier load balancing strategy visualized in the figures 10 and 11.

Cost of Co-mining: Mayura does not allocate CPU memory co-mining while allocating a relatively small amount of GPU memory (0.1%-2.5%) to offload contexts during inter-warp load-balancing. Mayura’s CPU backend replicates the baseline parallelization scheme and does not incur additional synchronization cost. However, the GPU backend needs to communicate an additional parameter (compared to the GPU baseline) with other threads in the warp to prevent duplicate exploration of sibling Nodes. This design decision allows *Mayura* to mine many more motifs simultaneously while incurring little to no memory or synchronization overhead.

Limiting sibling node exploration to a single thread for correctness underutilizes available parallelism across sibling nodes. While splitting the candidates and the search tree across threads could mitigate this, it may increase load balancing latency. Furthermore, GPU threads face constraints like register limits and reduced performance from control-flow divergence. The additional context and control logic (Algo. 3) required to enable co-mining thus become an overhead, potentially reducing performance gains. We can alleviate these drawbacks by minimizing the context and streamlining the control-flow by tuning the source code to the MG-Tree. §5 discusses optimization strategies to address these challenges.

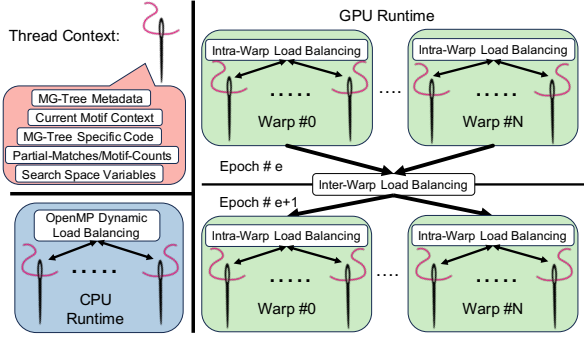


Figure 9: Visualizing Thread Context and Load-Balancing on the CPU and GPU at runtime.

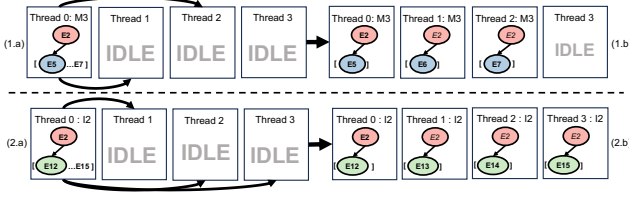


Figure 10: Intra-Warp Load-Balancing distributing candidates across idle warps. Only the source thread (i.e., 0) is allowed to explore sibling Nodes.

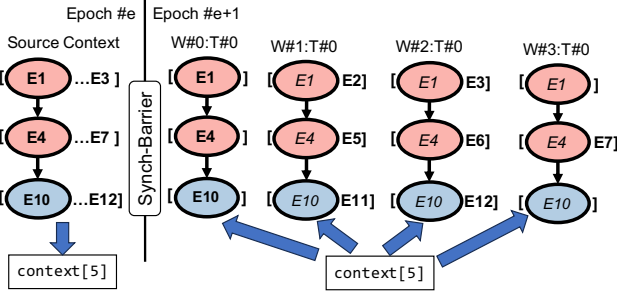


Figure 11: Inter-Warp Load-Balancing distributing candidates across warps from contexts dumped in the previous epoch.

5 MAYURA DESIGN OPTIMIZATIONS

The co-mining algorithm introduced in §4 achieves theoretical efficiency gains by exploiting structural and temporal commonalities across motifs via the MG-Tree. However, practical implementation on modern hardware architectures requires addressing critical performance bottlenecks unique to CPUs and GPUs. This section presents a suite of optimizations such as optimized code-generation and load-balancing that bridge the gap between algorithmic innovation and real-world execution efficiency.

5.1 Motif-Group Specific Code-generation

5.1.1 CPU Code-generation. For CPU implementations, we generate specialized loops for each level of the MG-Tree search hierarchy.

While the baseline implementation based on prior work [30, 57] uses recursive function calls with uniform loop structures, this approach confounds modern branch predictors due to varying iteration ranges across recursion levels. The code-generation phase for the CPU (Fig. 4, ④) unrolls the recursive search into distinct nested loops, each explicitly optimized for its corresponding MG-Tree level.

5.1.2 GPU Code-generation. GPUs present unique optimization challenges due to their Single Instruction Multiple Thread (SIMD) architecture and constrained register resources. Unlike CPUs that excel at handling complex control flow through speculative execution and sophisticated branch predictors, GPUs require fundamentally different optimizations to avoid performance pitfalls like warp divergence (threads in a warp executing different code paths) and register pressure (exceeding limited per-thread register capacity). Our GPU code generation strategy employs three synergistic optimizations to address these challenges while maintaining the algorithmic benefits of co-mining.

Register-Bound Context Mapping. Fig. 12(a) illustrates a portion of the code that enforces structural constraints by ensuring that the new candidate vertex, V , has not been matched before by comparing V with all vertices upto $V[mV-1]$, where mV is the number vertices that have been mapped and V is the array of vertices. We store these values (and the most of the context) in thread-local memory since the code has to flexibly work with any motif group with different number of vertices, edges, and motifs. This flexibility results in costing latency since the GPU incurs a memory operation to the shared or global memory. Given the MG-tree, we can replace the dynamic-array based context with fixed registers (e.g., $V0-V3$) like in Fig. 12(b), effectively hard-coding the mapping state for known motif sizes and eliminating memory accesses to the context.

In GPU architectures, warps execute instructions in lockstep across all 32 threads. Divergent control flow serializes execution, forcing subsets of threads to wait at synchronization points until all warp lanes complete their current path. The synchronization overhead, of tracking divergent paths and maintaining thread masks, incurs substantial latency, leading to a phenomenon known as warp divergence. While replacing loops with the switch-case in Fig. 12(b) reduces total branches, divergence persists when threads in a warp have different values for mV .

Predicated Control-Flow. Modern GPUs support predicated execution, where instruction execution is conditional on a Boolean value stored in predicate registers. This mechanism enables thread-specific control flow without explicit branching—instructions execute as no-ops (NOPs) when their associated predicate evaluates to false, thus eliminating warp-divergence and streamlining control-flow. As shown in Fig. 12(c), we utilize this feature to predicate the structural constraint checks based on the value of mV . The figure also contains two examples of predicated instructions in NVIDIA SASS, a low-level assembly language for NVIDIA GPUs. The first instruction is unconditionally executed since it is predicated on the always-true predicate PT, and the second one is dependent on a previously set predicate register P1. While predication avoids branch divergence, its application is limited by two constraints: 1) only arithmetic/logic operations can be predicated, and 2) predication is more effective with short code blocks, since longer predicated

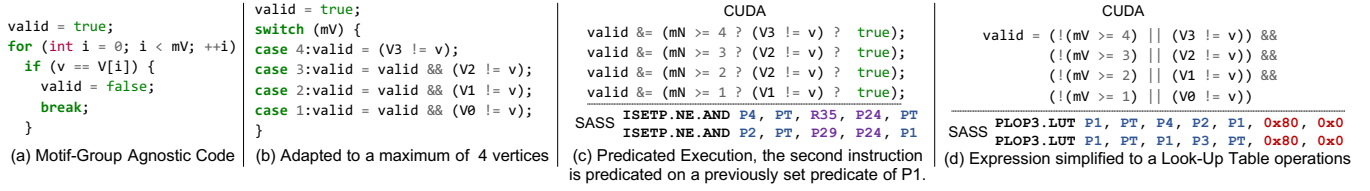


Figure 12: Optimization to reduce warp-divergence and streamline control-flow.



Figure 13: Sibling-Splitting for Intra-Warp Load-balancing.

blocks incur higher latencies by issuing instructions from both the if-then and else parts even when the threads do not diverge.

Expression Simplification. Rewriting multiple statements in Fig. 12(c) as a single boolean expression exposes further opportunities for the compiler. The predicated logic to test a specific vertex represents an implication from the value of mV to a check on the vertex, *i.e.*, $A \rightarrow B$, and can be simplified to a boolean expression resembling $\neg A \vee B$. These transformations enable the compiler to fuse multiple logical operations into 8-bit LookUp-Table (LUT) instructions, as illustrated in Fig. 12(d). This reduces the instruction count since multiple operations were replaced by a single LUT instruction and also reduces register pressure since registers are no longer required to hold as many intermediate values of an expression as before, thus improving occupancy.

In summary, by taking advantage of the compile-time knowledge of the maximum context size from the MG-Tree, and strategically applying optimizations across the code base, Mayura mitigates warp divergence, alleviates register pressure, and minimizes instruction counts. This synergy between algorithmic design and hardware awareness enables a more efficient exploitation of computational resources, yielding significant performance improvements.

5.2 GPU-Specific Load-balancing

While the two-tier load-balancing scheme effectively exploits parallelism across search-tree candidates, it under-utilizes parallelism available across Nodes in the MG-Tree. We address this limitation using two complementary optimizations: *sibling-splitting* for intra-warp parallelism and *multi-offloading* for inter-warp parallelism.

Sibling-Splitting for Intra-Warp Parallelism. Fig. 13 demonstrates the sibling-splitting optimization during intra-warp load balancing. The detection of idle threads triggers the load-balancing similar to §4.5 (Stage 1.a). For all the active threads, the system checks whether the Node being mined has an unexplored sibling according to the order determined in Algo. 2. For thread 0 mining M3 in Fig. 13, the unexplored sibling would be I2. Threads with such sibling Nodes (1) nominate an idle thread participating in the to mine the sibling (I2 \rightarrow thread 3), and (2) distributes the candidates of its search-tree across other threads (threads 0 - 2). Thread 3 then obtains the candidate list for I2 while retaining the search-tree for

I2's parent Node, I1. This approach enables the exploration of sibling Nodes in parallel, increasing the availability of candidates to reduce the number of idle threads.

Multi-Offload for Inter-Warp Parallelism is an optimization enabling concurrent exploration of Nodes across the MG-Tree hierarchy by decomposing the search context across multiple levels. Fig. 14 illustrates this approach during inter-warp load balancing, where a thread was mining for M3 with candidates for both edges in parent I1 before being interrupted to offload its context. Since siblings can be mined independent of each other, the optimization exposes this parallelism by creating separate contexts for candidates of each sibling (context[5] for M3, context[6] for I2) while preserving the search tree of their parent (I1). Since siblings are not part of the original search-tree, the candidate lists for a sibling is generated before its context is offloaded into global memory (*e.g.*, for I2 in Fig. 14). The same process is repeated for the parent Node and its siblings until the root Node is reached, with the search tree being trimmed to reflect the shallower depth in the MG-tree.

In the case of Fig. 14, a context is created that only contains I1's candidates (saved in context[8]). Note that all child Nodes have been explored with E4 a candidate for the second edge in I1, the I1-only context skips E4 and moves onto E5. This hierarchical decomposition exposes parallelism across as many levels as possible in the MG-tree with a given context by offloading multiple contexts for siblings in each level. When resuming mining in the next epoch in Fig. 14, the decomposed contexts are able to keep eight threads busy instead of just four threads without the optimization (Fig. 11). Observe that Sibling-Splitting paired with inter-warp load-balancing has a similar effect to that of Multi-Offload: by starting the search for siblings at an early stage, sibling-splitting creates multiple contexts from a single context, albeit constrained to the same level, and these multiple contexts are offloaded during load-balancing.

Additional Resource Footprint. While the above optimizations expose additional parallelism in the workload, they can introduce trade-offs that must be carefully managed. As intra-warp load balancing is invoked frequently, we constrain sibling-splitting to explore only one sibling at a time to minimize the overhead of exploring multiple siblings. The multi-offload strategy, while effective in distributing work, can incur additional instructions to offload multiple contexts. Our experiments reveal that these optimizations incur negligible overheads: dynamic instruction counts increase by $\leq 6\%$ and occupancy is reduced by $\leq 1\%$. These results confirm the practicality of our approach, as the performance benefits of enhanced parallelism outweigh the modest resource costs (§7).

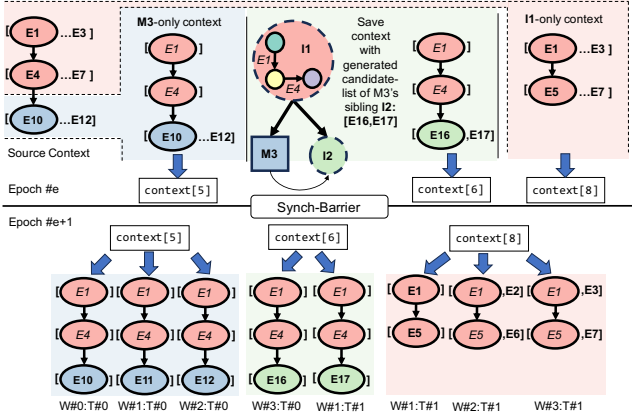


Figure 14: Optimization for Inter-Warp Load-balancing.

6 EVALUATION METHODOLOGY

The Baselines compared against Mayura’s CPU and GPU implementations respectively are the work proposed by Mackey et.al. [30] and Everest [57]. Both baseline methods exploit intra-query parallelism, by spreading out the search space for a single query across multiple threads.

The Hardware Setup consists of a server with an Intel Xeon Platinum 8380 CPU (40 cores, 80 threads) with 1TB of main memory, and an NVIDIA A40 GPU with 48GB GDDR6 memory.

Five Datasets of real-world temporal graphs spanning social networks, blockchain transactions, and internet traffic (Table 1) were used to evaluate Mayura. Since the equinix (eqx) dataset captures the exchange of internet packets between computers of two cities, making it a bipartite graph. Due to memory capacity limitations of our GPU, we subsample the massive eqx dataset to 37.5% of its original edges while preserving temporal characteristics.

Eight Queries were created by combining fourteen motifs, of which M1 - M11 are counted and M12 - M14 are only intermediates, which were used in prior work [24, 30, 38, 57], and are illustrated in Fig. 15. The queries cover three categories:

Depth-focused: Deepening MG-Trees (D1-D2).

Fanout-focused: Widening MG-Trees (F1-F3).

Complex Heterogenous: Variety of sizes and overlap (C1-C3).

To capture the notion of similarity among motifs, we define the Similarity Metric (SM) for a motif group MG and its MG-Tree as,

$$SM(MG, MG-Tree) = 1 - \frac{\sum_{M \in MG-Tree} (||E_M|| - ||E_{M.parent}||)}{\sum_{M \in MG} ||E_M||}$$

where $||E_M||$ is the number of edges in motif M . The denominator represents the aggregate edge count across all motifs in MG , while the numerator captures the cumulative incremental edge count relative to their parent Nodes in the $MG-Tree$. Higher inter-motif similarity reduces parent-child edge differentials, thereby decreasing the numerator and increasing SM values. Motif groups with elevated SM scores typically exhibit greater opportunities for computational reuse through our MG-Tree traversal. However, realized speedups remain contingent on system-specific factors including hardware utilization and load-balancing efficiency. Since the timescale of

Graph	#Vertices	#Temporal Edges	# Static Edges	Time Span	δ Window
wiki-talk (wt) [25]	1,140,149	7,833,140	2,787,968	6.24 years	1 day
stackoverflow (sxo) [25]	2,601,977	63,497,050	34,875,685	7.6 years	1 day
reddit-reply (trr) [26]	8,901,033	646,044,687	435,290,421	10.1 years	10 h
ethereum (eth) [23]	66,323,478	628,810,973	186,064,655	3.58 years	1 h
equinix (eqx) [38]	6,208,412	872,124,829	29,766,272	23.46 mins	3.6 ms

Table 1: Temporal graph datasets used for evaluation.

events varies across the datasets, we employ specific δ values that reflect meaningful time-windows and limit run-time [57].

7 EVALUATION RESULTS

High Level Summary: The evaluation results reveal substantial performance improvements from integrating co-mining techniques with our optimizations. Figures 16 and 17 capture the timings, in seconds, on the CPU and GPU respectively. For each query (title), the baseline and a set of co-mining optimizations (columns) are tested across multiple datasets (rows). Figures 18 and 19 capture the individual speedups for each query over the baseline for the corresponding set of optimizations on the CPU and GPU respectively, with the "Geomean" numbers being the geometric mean of the speedup across all queries for a particular dataset. On the CPU (Fig. 18), individual speedups range from 1.05 \times to 8.37 \times , with co-mining alone yielding an overall average improvement of 2.35 \times and code-generation pushing it up to an average of 2.46 \times . On the GPU (Fig. 19), individual speedups range from 0.82 \times to 7.59 \times , just using co-mining yields an overall average improvement of 1.48 \times with other optimizations raising it to 1.73 \times . While the absolute time saving for the experiments on the GPU, spanning shorter run-times (milliseconds to minutes), may appear modest at best, the speedups remain practically significant in high-throughput analytical environments where thousands of such queries are executed daily [22, 28, 51, 54]. By combining the multi-query processing approach with the intra-query parallelism implemented in the baselines, the overall process is more efficient when exploring the search space in parallel. These gains are strongly influenced by the degree of structural similarity among motifs: motif-groups with higher overlap benefit more from the co-mining strategy, whereas groups with minimal overlap (e.g., the C1 motif group) exhibit limited improvements and, in some cases, even a performance degradation on the GPU due to increased resource constraints. Note that mining C1 on the eth dataset on the CPU exceeded the time-limit of 24 hours and has been omitted from our comparisons.

Mayura effectively exploits structural similarities among motifs. D2 experiences higher speedups than D1 due to the implicit mining of M1 before mining M4, even if M1 is not counted explicitly. This trend is consistently observed when moving from F1 to F3 and C2 to C3, where the expansion of the MG-tree to include more structurally similar motifs correlates with improved speedups. C1, characterized by low inter-motif overlap (S.M.), proves challenging to optimize, resulting in the lowest speedups and even performance degradation on the GPU due to reduced occupancy and increased warp divergence compared to the baseline.

Dataset characteristics significantly influence performance gains. On the CPU, all datasets benefit from co-mining and most of them benefit from the code-generation as well. The bipartite

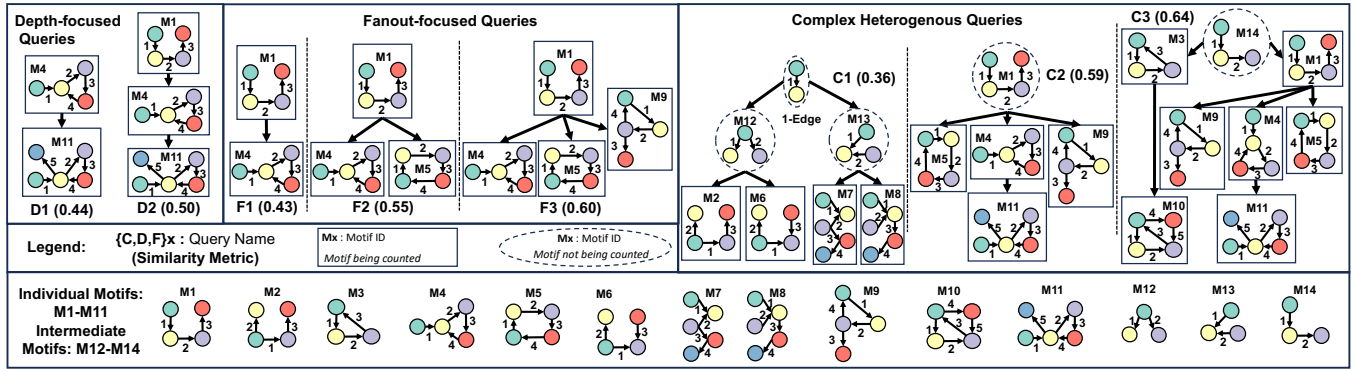


Figure 15: MG-Trees of Motif Groups, with respective (SM).

D1 (0.44)				D2 (0.5)				F1 (0.43)			
wtt	1.6	1.0	0.9	2.2	1.0	0.9	1.2	0.6	0.6	0.6	0.6
sxo	11.5	5.3	5.1	16.3	5.4	5.1	10.5	5.2	5.1	5.1	5.1
trr	708.0	498.0	461.0	832.0	502.0	461.0	320.0	194.0	180.0	180.0	180.0
eth	591.0	467.0	449.0	661.0	468.0	451.0	168.0	91.1	89.3	89.3	89.3
eqx	103.0	46.6	49.7	154.0	46.6	49.6	103.0	46.7	49.8	49.8	49.8
F2 (0.55)				F3 (0.6)				C1 (0.36)			
wtt	1.8	0.6	0.6	2.4	0.7	0.6	24.7	17.0	15.5	15.5	15.5
sxo	16.2	5.8	5.5	21.6	6.2	5.8	32.8	22.7	21.4	21.4	21.4
trr	499.0	225.0	207.0	660.0	249.0	229.0	44.7k	43.8k	42.6k	42.6k	42.6k
eth	264.0	115.0	112.0	362.0	143.0	139.0	TO	TO	TO	TO	TO
eqx	154.0	46.7	49.7	206.0	46.5	49.6	216.0	57.4	56.9	56.9	56.9
C2 (0.59)				C3 (0.64)				F2 (0.55)			
wtt	2.7	1.0	1.0	5.0	1.3	1.2	0.06	0.04	0.03	0.03	0.03
sxo	22.5	6.2	5.8	42.9	8.5	8.0	0.41	0.22	0.20	0.20	0.20
trr	1.05k	0.55k	0.51k	1.80k	0.86k	0.81k	33.10	23.90	17.80	17.70	17.70
eth	785.0	522.0	496.0	1.79k	1.24k	1.19k	20.30	22.30	15.90	15.80	15.40
eqx	206.0	46.7	49.3	411.0	46.7	49.1	0.57	0.28	0.28	0.28	0.28
C2 (0.59)				C3 (0.64)				F3 (0.6)			
wtt	0.18	0.17	0.17	0.17	0.15	0.12	0.06	0.04	0.04	0.03	0.03
sxo	0.84	0.49	0.45	0.44	0.44	0.44	0.51	0.22	0.20	0.20	0.20
trr	58.40	42.30	36.10	36.00	34.50	34.50	37.70	24.00	17.70	17.60	17.60
eth	24.50	23.80	20.80	19.50	17.60	17.60	21.10	22.40	15.90	15.90	15.30
eqx	1.14	0.28	0.29	0.29	0.28	0.28	0.85	0.28	0.28	0.28	0.28
C2 (0.59)				C3 (0.64)				C1 (0.36)			
wtt	0.25	0.25	0.21	0.19	0.18	0.18	0.04	0.03	0.03	0.02	0.02
sxo	1.38	0.72	0.65	0.65	0.65	0.65	0.30	0.22	0.20	0.20	0.20
trr	91.40	65.70	58.30	58.00	55.90	55.90	18.70	14.70	13.40	13.40	13.40
eth	51.00	58.90	58.80	53.00	50.50	50.50	3.15	2.82	2.65	2.45	2.38
eqx	2.27	0.30	0.30	0.30	0.30	0.30	0.57	0.28	0.29	0.29	0.29

Figure 16: CPU Timings (seconds).

Figure 17: GPU Timings (seconds).

D1 (0.44)		D2 (0.5)		F1 (0.43)		
wtt	1.52	1.70	2.27	2.39	2.13	2.12
sxo	2.17	2.25	3.04	3.20	2.00	2.08
trr	1.42	1.54	1.66	1.80	1.65	1.78
eth	1.27	1.32	1.41	1.47	1.84	1.88
eqx	2.21	2.07	3.30	3.10	2.21	2.07
F2 (0.55)		F3 (0.6)		C1 (0.36)		
wtt	2.97	3.03	3.61	3.75	1.45	1.59
sxo	2.78	2.95	3.48	3.74	1.44	1.53
trr	2.22	2.41	2.65	2.88	1.02	1.05
eth	2.30	2.36	2.53	2.60	TO	TO
eqx	3.30	3.10	4.43	4.15	3.76	3.80
C2 (0.59)		C3 (0.64)		Geomean		
wtt	2.66	2.71	3.91	4.04	2.42	2.53
sxo	3.64	3.85	5.06	5.38	2.77	2.92
trr	1.91	2.05	2.09	2.24	1.76	1.89
eth	1.50	1.58	1.44	1.50	1.70	1.76
eqx	4.41	4.18	8.80	8.37	3.69	3.50
c		cg		c		cg

Legend: c: co-mining; cg: [c] + motif-specific code-gen.

TO: Time Out

Legend: c: co-mining; cg: [c] + motif-specific code-gen; TO: Time Out

D1 (0.44)					D2 (0.5)					F1 (0.43)				
wtt	1.35	1.48	1.74	1.81	1.64	1.79	2.19	2.20	1.19	1.31	1.62	1.62		
sxo	1.87	2.02	2.03	2.03	2.28	2.49	2.50	2.50	1.38	1.47	1.48	1.48		
trr	1.38	1.86	1.87	1.87	1.57	2.13	2.14	2.14	1.27	1.40	1.40	1.40		
eth	0.91	1.28	1.28	1.32	0.94	1.33	1.33	1.38	1.12	1.19	1.29	1.32		
eqx	2.06	2.00	2.01	2.01	3.09	3.01	3.01	3.01	2.06	1.99	1.99	1.99		
F2 (0.55)					F3 (0.6)					C1 (0.36)				
wtt	1.34	1.59	1.72	1.74	0.98	1.00	1.12	1.44	0.64	0.74	0.79	0.82		
sxo	1.39	1.54	1.54	1.54	1.51	1.59	1.62	1.62	0.94	0.95	0.95	0.95		
trr	1.31	1.44	1.44	1.44	1.38	1.46	1.47	1.51	0.88	0.98	0.98	0.98		
eth	1.18	1.30	1.36	1.39	1.21	1.25	1.32	1.42	0.85	1.03	1.04	1.05		
eqx	3.07	2.98	2.98	2.98	4.06	3.92	3.92	3.93	1.76	1.69	1.69	1.71		
C2 (0.59)					C3 (0.64)					Geomean				
wtt	1.05	1.08	1.17	1.54	0.98	1.19	1.27	1.34	1.11	1.23	1.39	1.51		
sxo	1.72	1.87	1.92	1.92	1.91	2.11	2.13	2.13	1.58	1.69	1.71	1.71		
trr	1.38	1.62	1.62	1.69	1.39	1.57	1.58	1.64	1.31	1.52	1.53	1.55		
eth	1.03	1.18	1.26	1.39	0.87	0.87	0.96	1.01	1.01	1.17	1.22	1.28		
eqx	4.05	3.97	3.97	4.00	7.69	7.57	7.57	7.59	3.12	3.03	3.03	3.04		
	c	cg	cgs	cgsm	c	cg	cgs	cgsm	c	cg	cgs	cgsm		

Legend: c: co-mining; cg: [c] + motif-specific code-gen; cgs: [cg] + sibling-splitting; cgsm: [cgs] + multi-offload

Legend: c: co-mining; cg: [c] + motif-specific code-gen; cgs: [cg] + sibling-splitting; cgsm: [cgs] + multi-offload

Figure 18: Breakdown of performance improvements of different optimizations on the CPU, compared to the baseline.

Figure 19: Breakdown of performance improvements of different optimizations on the GPU, compared to the baseline.

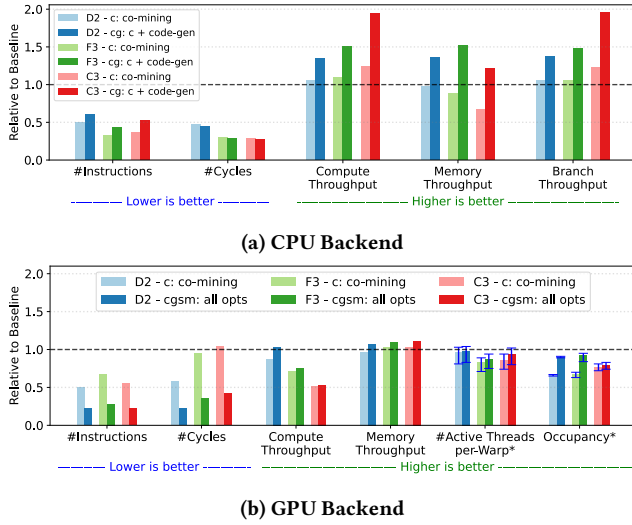


Figure 20: Architectural Metrics for mining D2, F3 and C3 on wtt, relative to the baseline.

eqx dataset exhibits exceptionally high speedups on both CPU and GPU platforms, due to the fact that bipartite graph cannot allow motifs that connect vertices in the same disjoint partition, naturally eliminating any match. Consider D1, where M1 can be found in a bipartite graph since $(\text{blue}, \text{purple}) \in \text{partition } \#0$ and $(\text{yellow}, \text{red}) \in \text{partition } \#1$. Proceeding to M4 after matching M1, we would fail to find any matches since there are no edges between yellow and red as they $\in \text{partition } \#1$. This in-turn prunes the search for any descendants of M4, i.e., M11, since they depend on matching M4 first. This way, the algorithm is able to eliminate multiple redundant searches required for M4 and M11 when mining them individually. wtt, trr, and eth datasets benefit from all GPU optimizations, although the extent varies by motif-group. sxo’s performance peaks at the cgs optimization, suggesting a well-balanced workload distribution that obviates the need for inter-warp load balancing. The reduced speedups observed for eth and trr across all queries stems from their elevated motif match density (i.e., $\sigma = ||\text{matches}|| \div ||E_G||$). With a higher σ , threads process significantly larger candidate sets per edge, prolonging exploration of individual MG-Tree Nodes and delaying transitions to sibling Nodes. This increases the serialization of the search across the MG-tree hierarchy, reducing the efficacy of co-mining. Conversely, eqx achieves superior performance due to its bipartite structure leading to the smallest σ among all datasets, enabling aggressive pruning of the search space.

Performance Analysis: To explain the underlying factors contributing to these performance improvements, we analyzed key architectural metrics for both CPU (Fig. 20a) and GPU (Fig. 20b) implementations, which have been collected by running the queries for motif groups D2, F3 and C3 on wtt. The figures compare five architectural metrics of performing only co-mining and co-mining with optimizations, with the baselines of the respective implementation. The “#Active Threads per Warp” and “Occupancy” metrics for the GPU baseline were computed as time-weighted averages across the individual baseline kernels. This approach accounts for the varying

occupancy characteristics and performance profiles of the baseline kernels. The error bars indicate the extent to which the optimized kernels outperform / underperform the baseline kernels. Using standalone CPU co-mining or combining it with code-generation effectively reduces the number of instructions executed, an indication that the system overall is performing less work. The efficacy of code-generation to address branch prediction is reflected in the higher branch-instruction throughput (20a). While GPU co-mining reduces instruction counts, its efficacy diminishes with larger motif groups due to reduced occupancy and increased warp divergence. Our optimizations, aimed at streamlining control flow and eliminating unnecessary instructions, partially mitigate these issues by increasing the average number of active threads per warp. However, the complex control flow inherent in co-mining constrains the overall effectiveness of multi-offload and sibling-splitting optimizations. We also performed an experiment to evaluate the efficacy of improving the occupancy at the cost of increased memory operations. We chose C3 motif group since its kernel exhibits the lowest occupancy (44%) among all motif groups due to high register usage for maintaining motif counts. Offloading counters to thread-local memory increased occupancy to 70% but yielded only marginal performance improvements due to increased memory accesses, underscoring the trade-off between occupancy and memory access efficiency in GPU implementations.

Memory Footprint: Although we do not allocate any extra global memory explicitly for the CPU implementation, the additional footprint has ranged between 100KB less to 180KB over the footprint of CPU baseline mining only one motif. We dismiss this as noise, especially when the smallest dataset has a footprint of 3GB in CPU RAM. That said, the GPU backend does allocate extra global memory to enable the inter-warp load-balancing. This is due to the extra context needed to guide the search in the case of co-mining, as opposed to single motif mining. This extra space turns out to be entirely dependent on the motif group: 14MB for F3, 16MB for D2, 20MB for D3. Each motif group has different requirements since it’s thread-local context would scale with longer motifs or wider MG-Tree. These overheads are still relatively small when comparing the smallest of wtt at 800MB (2.5% overhead) to the the largest dataset of trr at 17Gb (0.1%). Also, the GPU baseline and Mayura’s GPU backend use a more space efficient graph representation for the GPU (800MB for wtt) than CPU (3GB for wtt) to mitigate the space constraints of GPUs. This design decision allows Mayura to process many more motifs concurrently while maintaining a similar footprint.

GPU Footprint: The larger context held within a GPU thread to enable co-mining reduces the amount of parallelism available to exploit, with the limited register file acting as a bottleneck on the number of active threads/blocks. As shown in the Table 2, this increased register requirement reduces the number of active GPU blocks, effectively reducing the number of threads that can execute simultaneously, and creating a performance trade-off as we scale the number of co-mined motifs.

Effect of δ : We evaluated the efficacy of Mayura under varying temporal constraints, by comparing its performance to the baseline, mining D2, F3, and C3 on the wtt dataset, with time-window settings of $\delta/2$, δ , and $2 * \delta$. The results in Figs. 21a and 21b indicate that shorter time-windows lead to a greater speedup relative to the

#Motifs	#Registers	#Blocks	Reduction in #Blocks
1	44	1092	0%
4	55	1008	8%
8	67	756	31%

Table 2: Impact of co-mining on GPU register utilization and thread occupancy

```

1 if is_bipartite(data_graph):
2     co_mining = True
3 elif platform == "GPU" and SM < 0.44:
4     co_mining = False
5 else:
6     choose_smaller(delta)
7     co_mining = True

```

Listing 1: Heuristic for Co-Mining

baseline on both the CPU and GPU. Longer time windows expand the candidate set and widen the search tree, potentially across multiple levels. Such wide search trees lead to load-imbalances in the system when they are not split-up across multiple compute units. This is particularly evident with the CPU backend since it performs balancing only at the top-level. Whereas the GPU implementation, which employs finer-grained load-balancing across all levels in the search-tree, is less adversely affected.

Heuristic for Co-Mining: The efficacy of co-mining for a given motif group and dataset combination can be anticipated by analyzing the structure of the graph, S.M and δ . Co-mining on a bipartite graph has always resulted in a performance improvement on both platforms, as it disallows motif matches to have edges incident within same partition. For co-mining to be more efficient than the GPU baseline, it needs a minimum S.M. to offset the tighter architectural constraints (Tab. 2), which we found to be 0.44 from our evaluation. A user with flexibility to choose δ could reduce the time-window to improve performance (Fig. 21). Based on our evaluation results, we propose a heuristic (Lst. 1) that determines whether co-mining would be beneficial.

8 RELATED WORK

Multi-Query Optimization (MQO): Mayura addresses the unique challenges of MQO in the context of temporal motif mining. In the past, GEQO [16] pioneered ML-based identification of semantically equivalent subexpressions, while Ma *et al.* [29] extended MQO to continuous subgraph matching in dynamic graphs. MapReduce adaptations [53] demonstrated MQO’s versatility across paradigms. While these approaches focus on relational queries or static graph processing, Mayura extends the concept of multi-query optimization to the domain of temporal graph mining, identifying structural and temporal commonalities across multiple motifs.

Static Graph Mining systems count matches of a pattern based on the structure of the query pattern [11, 18, 19, 31, 32, 42, 45, 46, 56]. Notable contributions include Arabesque [48], which introduced a distributed framework for graph mining, and Peregrine [18, 19], which optimized pattern-aware exploration. G2Miner [11] further advanced the field by synthesizing pattern-specific code for GPUs, similar to Everest and Mayura. While these systems have advanced

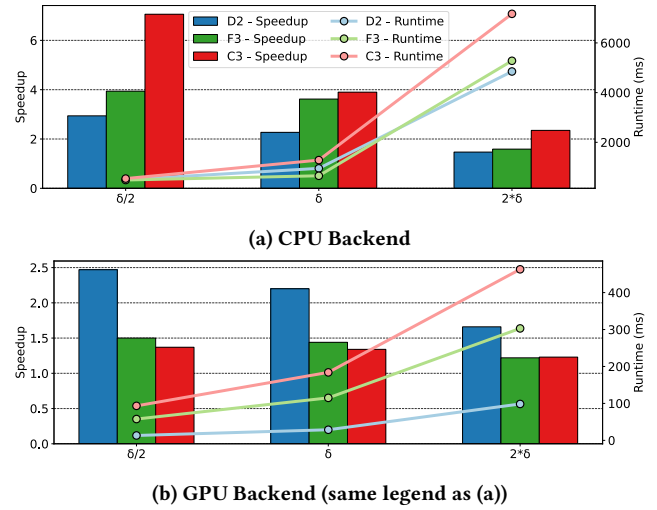


Figure 21: Effect of scaling δ on Speedup and Runtime.

the state-of-the-art in static graph mining, they do not address the unique challenges posed by temporal constraints in motif mining. **Temporal Motif Mining** introduces additional complexity by incorporating temporal ordering constraints within a specified time window. Prior work in this field falls into two categories: (1) *exact methods* like Mackey’s chronological edge-matching [30] and Everest’s GPU acceleration [57], which we extend through co-mining; (2) *approximate techniques* estimate the number of matches with high accuracy, significantly reducing computation time from days to minutes [33]. They estimate the number of matches by either sampling a subset of edges [55], paths [33] or time intervals [27, 37, 38]. Oden [37] can estimate multiple motifs with the same underlying structure. Despite these advancements, most existing systems are limited to CPU-based implementations [13, 33], are optimized for only a few motifs [13] or trade accuracy for performance [27, 37, 37, 38, 55]. Mayura provides a flexible framework capable of handling a wide range of motifs, co-mining them efficiently across CPUs and GPUs, without compromising accuracy.

Hierarchical Indices: FERRARI [52] and PRAGUE [21] also exploit hierarchical structures like the MG-Tree, but their goals and designs differ fundamentally from Mayura. Both FERRARI’s ADVISE and PRAGUE’s SPIG indexes are built on-the-fly during visual query formulation to record matches of fragments of a single evolving query, and to guide incremental similarity search. In contrast, the MG-Tree is an offline, compile-time hierarchy over multiple distinct temporal motifs submitted together, grouping them by common edge prefixes (both structural and temporal) to share search paths across all motifs. While ADVISE and SPIG could accomplish the task of the MG-Tree by building the index with static subgraphs of the motifs and then filtering them for temporal constraints, the need for ADVISE and SPIG to enumerate their candidates for each fragment of the evolving query results in a large memory and computational overhead. These overheads are in addition to the computational overhead of enumerating structurally compliant matches before filtering them for temporal constraints (§2.1). The MG-Tree does not suffer from these computational and memory

overheads as it does not enumerate all partial matches, and expands them only when temporal and structural constraints are met.

9 CONCLUSION

Mayura addresses the critical challenge of efficiently mining multiple temporal motifs by introducing a novel co-mining paradigm that exploits structural and temporal similarities across query patterns. Our framework introduces the Motif-Group Tree (MG-Tree), a hierarchical data structure that systematically organizes motifs. Experimental results demonstrate significant performance improvements, with overall speedup of $2.5\times$ on the CPU and $1.7\times$ on the GPU across diverse datasets. The effectiveness of our approach hinges on the MG-tree exploiting motif similarity to reduce redundant work and exploiting parallelism in the workload. Architectural bottlenecks posed by enabling co-mining were mitigated by optimizing code-generation. These advancements not only enhance temporal graph analytics but also underscore the importance of hardware-aware co-design for scalable motif mining.

REFERENCES

- [1] 2025. CUDA Handbook. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/contents.html>
- [2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*. 29–42.
- [3] Erik Altman, Jovan Blanuša, Luc Von Niederhäusern, Béni Egressy, Andreea Anghel, and Kubilay Atas. 2023. Realistic synthetic financial transactions for anti-money laundering models. *Advances in Neural Information Processing Systems* 36 (2023), 29851–29874.
- [4] Subi Arumugam, Alin Dobra, Christopher M Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 519–530.
- [5] Shilpa Balan and Janhavi Rege. 2017. Mining for social Media: Usage patterns of small businesses. *Business Systems Research Journal* 8, 1 (March 2017), 43–50. <https://doi.org/10.1515/bsrj-2017-0004>
- [6] Jovan Blanuša, Maximo Cravero Baraja, Andreea Anghel, Luc Von Niederhäusern, Erik Altman, Haris Pozidis, and Kubilay Atas. 2024. Graph Feature Preprocessor: Real-time Subgraph-based Feature Extraction for Financial Crime Detection. In *Proceedings of the 5th ACM International Conference on AI in Finance*. 222–230.
- [7] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. 2006. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 479–490.
- [8] Jianhong Cai, Hao Gu, and Xiaorong Zhu. 2023. Mobility-Aware Offloading Scheme for 6G’s Real-Time Tasks with Temporal Graphs and Graph Matching. *2023 International Conference on Networks, Communications and Intelligent Computing (NCIC)* (2023), 189–194. <https://api.semanticscholar.org/CorpusID:269806405>
- [9] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*.
- [10] Ming-Syan Chen, Jiawei Han, and P.S. Yu. 1996. Data mining: an overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering* 8, 6 (1996), 866–883. <https://doi.org/10.1109/69.553155>
- [11] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 857–877. <https://www.usenix.org/conference/osdi22/presentation/chen>
- [12] Lawrence Fisher. 2024. Leveraging Graph Databases for Fraud Detection in Financial Systems. *Communications of ACM* (Oct. 2024). <https://cacm.acm.org/blogcacm/leveraging-graph-databases-for-fraud-detection-in-financial-systems/>
- [13] Z. Gao, C. Cheng, Y. Yu, L. Cao, C. Huang, and J. Dong. 2022. Scalable Motif Counting for Large-scale Temporal Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2656–2668. <https://doi.org/10.1109/ICDE53745.2022.00244>
- [14] Joshua Glasser and Brian Lindauer. 2013. Bridging the gap: A pragmatic approach to generating insider threat data. In *2013 IEEE Security and Privacy Workshops*. IEEE, 98–104.
- [15] László Hajdu and Miklós Krész. 2020. Temporal network analytics for fraud detection in the banking sector. In *ADBIS, TPD and EDA 2020 Common Workshops and Doctoral Consortium*. Springer, 145–157.
- [16] Brandon Haynes, Rana Alotaibi, Anna Pavlenko, Jyoti Leeka, Alekh Jindal, and Yuanyuan Tian. 2023. GEQO: ML-Accelerated Semantic Equivalence Detection. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.
- [17] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).
- [18] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. , Article 13 (2020), 16 pages. <https://doi.org/10.1145/3342195.3387548>
- [19] Kasra Jamshidi and Keval Vora. 2021. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 1–10. <https://doi.org/10.1145/3469379.3469381>
- [20] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating graph mining systems with subgraph morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 162–181.
- [21] Changjiu Jin, Sourav S Bhowmick, Byron Choi, and Shuigeng Zhou. 2012. Prague: towards blending practical visual subgraph query formulation and query processing. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 222–233.
- [22] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.
- [23] Dániel Kondor, Nikola Bulatovic, József Stéger, István Csabai, and Gábor Vattay. 2021. *Ethereum Transaction Network*. <https://doi.org/10.5281/zenodo.4543269> Data used in our upcoming paper: Kondor D, Bulatovic N, Stéger J, Csabai I, Vattay G (2021). The rich still get richer: Empirical comparison of preferential attachment via linking statistics in Bitcoin and Ethereum. Under review. <https://arxiv.org/abs/2102.12064>.
- [24] Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, and Panayiotis Tsparas. 2018. Flow Motifs in Interaction Networks. In *International Conference on Extending Database Technology*.
- [25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [26] Paul Liu, Austin Benson, and Moses Charikar. 2018. A sampling framework for counting temporal motifs. *arXiv preprint arXiv:1810.00980* (2018).
- [27] Paul Liu, Austin R Benson, and Moses Charikar. 2019. Sampling methods for counting temporal motifs. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 294–302.
- [28] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.
- [29] Ziyi Ma, Jianye Yang, Xu Zhou, Guoqing Xiao, Jianhua Wang, Liang Yang, Kenli Li, and Xuemin Lin. 2024. Efficient Multi-Query Oriented Continuous Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3230–3243.
- [30] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE international conference on big data (big data)*. IEEE, 3972–3979.
- [31] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, , and B. Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. In *arXiv preprint arXiv:1911.12877*.
- [32] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. (2019), 509–523. <https://doi.org/10.1145/3341301.3359633>
- [33] Yunjie Pan, Omkar Bhalerao, C Seshadhri, and Nishil Talati. 2024. Accurate and Fast Estimation of Temporal Motifs using Path Sampling. *arXiv preprint arXiv:2409.08975* (2024).
- [34] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.
- [35] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment* 10, 3 (2016), 121–132.
- [36] Ilie Sarpe and Fabio Vandin. 2021. OdeN: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1568–1577.
- [37] Ilie Sarpe and Fabio Vandin. 2021. OdeN: Simultaneous Approximation of Multiple Motif Counts in Large Temporal Networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (Virtual Event, Queensland, Australia) (CIKM ’21)*. Association for Computing Machinery, New York, NY, USA, 1568–1577. <https://doi.org/10.1145/3459637.3482459>
- [38] Ilie Sarpe and Fabio Vandin. 2021. PRESTO: Simple and Scalable Sampling Techniques for the Rigorous Approximation of Temporal Motif Counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 145–153.

- [39] Timos Sellis and Subrata Ghosh. 1990. On the multiple-query optimization problem. *IEEE Transactions on Knowledge & Data Engineering* 2, 02 (1990), 262–266.
- [40] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.
- [41] Huijuan Shao, Manish Marwah, and Naren Ramakrishnan. 2013. A temporal motif mining approach to unsupervised energy disaggregation: Applications to residential and commercial buildings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 27. 1327–1333.
- [42] T. Shi, M. Zhai, Y. Xu, and J. Zhai. 2020. *GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination*. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.
- [43] Floyd Smith. 2023. Case study: Fraud detection “On the swipe” for a major US bank. <https://www.singlestore.com/blog/case-study-fraud-detection-on-the-swipe/>
- [44] Corey Sommers. 2024. Advanced Fraud Detection in Financial Services | ArangoDB. <https://arangodb.com/2024/03/advanced-fraud-detection-in-financial-services-with-arangodb-and-aql/>
- [45] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapid-Match: a holistic approach to subgraph query processing. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 176–188. <https://doi.org/10.14778/3425879.3425888>
- [46] Xibo Sun and Qiong Luo. 2023. Efficient gpu-accelerated subgraph matching. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.
- [47] Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. 2019. Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network. *IEEE Access* 7 (2019), 49778–49789.
- [48] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulhaga. 2015. Arabesque: A System for Distributed Graph Mining. (2015), 425–440. <https://doi.org/10.1145/2815400.2815410>
- [49] TigerGraph. 2022. Anti-Money Laundering with Graph DB | TigerGraph. <https://www.tigergraph.com/solutions/anti-money-laundering-aml/>
- [50] Yicheng Tu, Mehrad Eslami, Zichen Xu, and Hadi Charkhgard. 2022. Multi-Query Optimization Revisited: A Full-Query Algebraic Method. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 252–261.
- [51] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706.
- [52] Chaohui Wang, Miao Xie, Sourav S Bhowmick, Byron Choi, Xiaokui Xiao, and Shuigeng Zhou. 2020. FERRARI: an efficient framework for visual exploratory subgraph search in graph databases. *The VLDB Journal* 29 (2020), 973–998.
- [53] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.
- [54] Jiaqi Wang, Tianyi Li, Anni Wang, Xiaoze Liu, Lu Chen, Jie Chen, Jianye Liu, Junyang Wu, Feifei Li, and Yunjun Gao. 2023. Real-time workload pattern analysis for large-scale cloud databases. *arXiv preprint arXiv:2307.02626* (2023).
- [55] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2020. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1505–1514.
- [56] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Dallas, Texas) (SC ’22)*. IEEE Press, Article 53, 13 pages.
- [57] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2023. Everest: GPU-Accelerated System for Mining Temporal Motifs. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 162–174. <https://doi.org/10.14778/3626292.3626299>
- [58] Cécile Zachlod, Olga Samuel, Andrea Ochsner, and Sarah Werthmüller. 2022. Analytics of social media data – State of characteristics and application. *Journal of Business Research* 144 (2022), 1064–1076. <https://doi.org/10.1016/j.jbusres.2022.02.016>

A DESIGN

A.1 Definition of MG-Tree

For a group of temporal motifs $MG = \{M_1, M_2, \dots, M_k\}$, the MG-Tree MGT is defined as a hierarchical tree of Nodes that capture the similarities among motifs in MG , rooted at N_{root} .

Node Composition: Any Node $N \in MGT$ is composed of the following 3 members: C_N , $Children(N)$ and Q_N ,

$$N = \langle C_N, Children(N), Q_N \rangle, \text{ where,}$$

C_N :- The (common) motif with edges common across all descendants $C_{N_{desc}}$, effectively becoming a prefix for $C_{N_{desc}}$ with their first $|C_N|$ being equivalent to C_N .

$$\text{prefix}(C_{N_{desc}}, |C_N|) = C_N$$

$Children(N)$:- The set of immediate descendants (N_{child}) that are constructed by directly extending C_N , where C_N is the longest prefix of $C_{N_{child}}$, not including $C_{N_{child}}$.

$$C_N <_{\max} C_{N_{child}}$$

Q_N :- The reference to a query motif $M_i \in MG$ when C_N is equivalent to M_i , else is \emptyset . For all motifs $M_i \in MG$, there is exactly one Node N in MGT which is responsible for mining M_i , i.e. $Q_N = M_i$. Consequently, the union of all nonempty query motif references in MGT is the motif group,

$$MG = \bigcup Q_N \forall N \in MGT \mid Q_N \neq \emptyset$$

Root Node:- $N_{root} \in MGT$ whose common motif $C_{N_{root}}$ has edges common across all motifs in MG .

A.2 GPU Load-balancing

Intra-thread Synchronization operations on GPUs are relatively high-latency operations [1], which makes it expensive to monitor threads and calculate a global or even a local balance factor. Instead, Mayurasimplys this operation by monitoring the status of threads within a warp periodically (say every $INTRA_INTRVL$ iterations). At the end of a period, all threads within a warp vote to perform intra-warp load-balancing if any of the threads are idle and some have work to share. Monitoring the status of all warps across the GPU can only be done through the global memory, as it is the only piece of memory accessible to all threads across the GPU. To this end, a globally accessible byte of memory is set when a warp is entirely idle. Similar to intra-warp balancing, inter-thread balancing is gate-kepted to be performed at a certain interval. However, as global memory accesses are a lot more expensive than warp-level synchronization [1], the period to monitor warps for idleness is a lot longer ($INTER_INTRVL > INTRA_INTRVL$). Each thread then triggers inter-warp load-balancing when the globally accessible byte of memory is set to indicate idleness. The pseudocode in Listing 2 captures the logic of the two-tier load-balancing.

A.3 Code-Generation

Listing 3 illustrates the C++ code generated to mine the MG-Tree in Fig. 8.

```

1 global_idle = False
2 ...
3 while True:
4     loop_cnt += 1
5     if i < len(cand_list):
6         thread_idle = False
7         # Mining
8         edge = cand_list[i++]
9         ...
10    else:
11        thread_idle = True
12
13    if loop_cnt % INTRA_INTRVL && ANY(thread_idle):
14        if ALL(thread_idle):
15            global_idle = True
16        else:
17            # Perform Intra-Warp Load-Balancing
18
19    if loop_cnt % INTER_INTRVL && global_idle:
20        # Perform Inter-Warp Load-Balancing

```

Listing 2: Pseudocode for GPU-Load Balancing

```

1 for (Edge e1 : G) { // 1st edge in I1
2     list<Edge> cand2_I1 = ...;
3     for (Edge e2 : cand2_I1) { // 2nd edge in I1
4         list<Edge> cand3_M3 = ...;
5         for (Edge e3 : cand3) // 3rd edge in M3
6             if (M3.matches({e1, e2, e3, e4}))
7                 ++count_M3;
8         list<Edge> cand3_I2 = ...;
9         for (Edge e3 : cand3_I2) { // 3rd edge in I2
10            list<Edge> cand4_M4 = ...;
11            for (Edge e4 : cand4_M4) // 4th Edge in M4
12                if (M4.matches({e1, e2, e3, e4}))
13                    ++count_M4;
14            list<Edge> cand4_M5 = ...;
15            for (Edge e4 : cand4_M5) // 4th Edge in M5
16                if (M5.matches({e1, e2, e3, e4}))
17                    ++count_M5;
18        }
19    }
20 }

```

Listing 3: C++ code generated to mine MG-Tree in Fig. 8.

B EVALUATION RESULTS

B.1 Effect of δ

To better understand the effect of the size of the temporal window size, δ , on the performance, we expanded this evaluation by,

- (1) Adding four additional δ configurations: $\delta/4$, $\delta/3$, 3δ , and 4δ .
- (2) Incorporating two other datasets, Stack Overflow (sxo) and Temporal-Reddit-Reply (trr), alongside the original Wikitalk (wtt) dataset.

Our expanded results (Figures 22, 23, 24) reinforces the observations from the smaller scale experiment in the paper: Increasing the time window (δ) diminishes the performance advantage of co-mining over the baseline. This is because larger window sizes expand the candidate search space across all levels of the MG-Tree hierarchy, and sequentializing the exploration until the system can re-balance the load. While the GPU backend is able to load-balance

across all levels of the MG-Tree, the CPU backend is only able to do so at the top-most level, leading to a larger loss in the performance gap between the baseline and the co-mining execution. This is evident from the ration between the speedup achieved at $\delta/4$ and 4δ in Table 3.

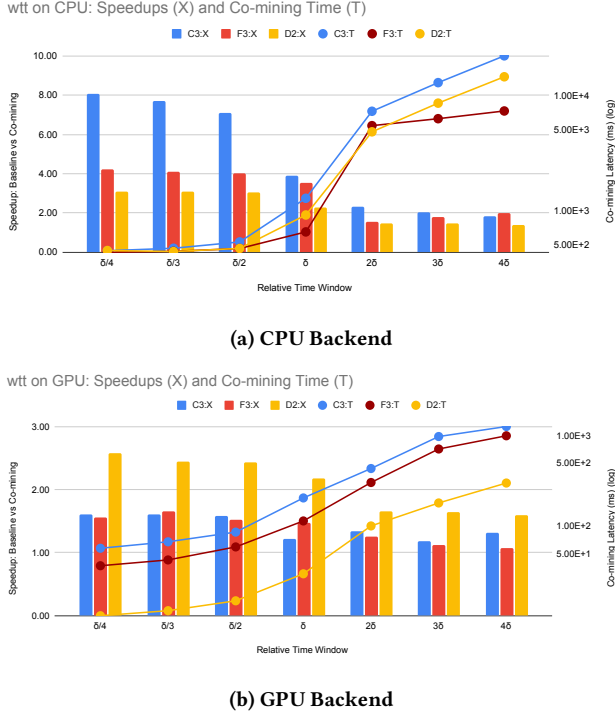


Figure 22: Effect of scaling δ on Speedup and Runtime on the Wikitalk (wtt) dataset.

MG	GPU Ratio	CPU Ratio
D2	1.62	2.22
F3	1.54	2.71
C3	1.36	4.42

Wikitalk (wtt)

MG	GPU Ratio	CPU Ratio
D2	1.62	2.22
F3	1.54	2.71
C3	1.36	4.42

Stack-Overflow (sxo)

MG	GPU Ratio	CPU Ratio
D2	1.62	2.22
F3	1.54	2.71
C3	1.36	4.42

Temporal Reddit Reply (trr)

Table 3: Ratio between the speedup at $\delta/4$ vs 4δ for different datasets.

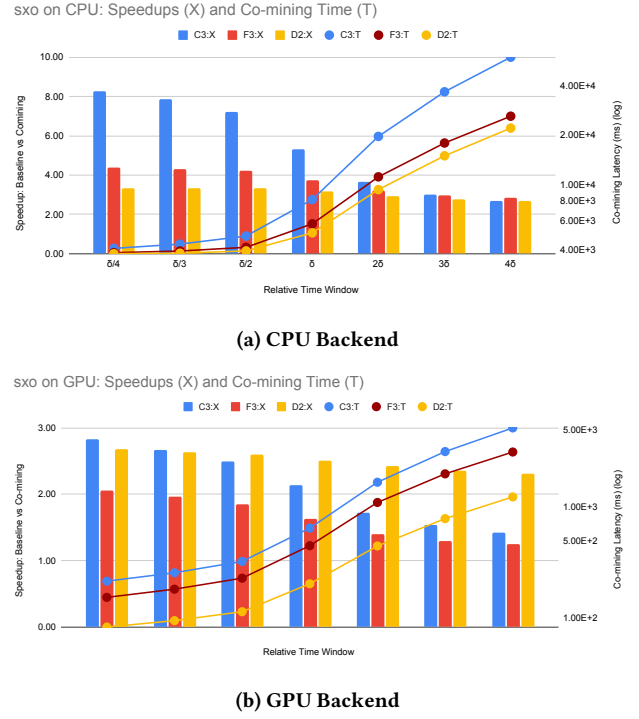
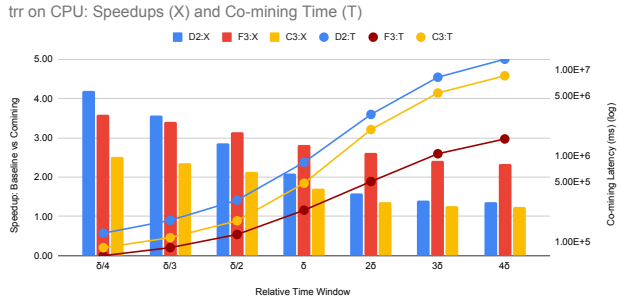
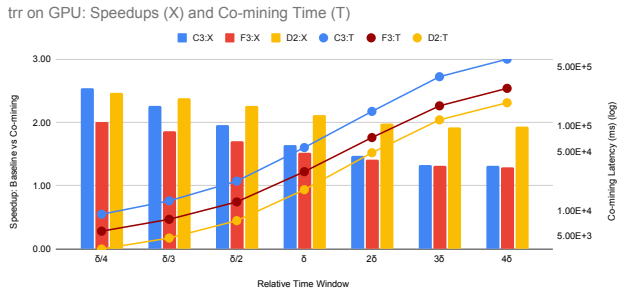


Figure 23: Effect of scaling δ on Speedup and Runtime on the Stack Overflow (sxo) dataset.



(a) CPU Backend



(b) GPU Backend

Figure 24: Effect of scaling δ on Speedup and Runtime on the trr dataset.