Mint: An Accelerator For Mining Temporal Motifs

Nishil Talati*, Haojie Ye*, Sanketh Vedula†, Kuan-Yu Chen‡, Yuhan Chen*, Daniel Liu*, Yichao Yuan*,
David Blaauw‡, Alex Bronstein†, Trevor Mudge*, Ronald Dreslinski*
*Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA
†Computer Science, Technion, Haifa, Israel
‡Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI, USA
Email: talatin@umich.edu

***Abstract*—A variety of complex systems, including social and communication networks, financial markets, biology, and neuroscience are modeled using temporal graphs that contain a set of nodes and directed timestamped edges. Temporal motifs in temporal graphs are generalized from subgraph patterns in static graphs in that they also account for edge ordering and time duration, in addition to the graph structure. Mining temporal motifs is a fundamental problem used in several application domains. However, existing software frameworks offer sub-optimal performance due to high algorithmic complexity and irregular memory accesses of temporal motif mining.**

This paper presents Mint—a novel accelerator architecture and a programming model for mining temporal motifs efficiently. We first divide this workload into three fundamental tasks: search, book-keeping, and backtracking. Based on this, we propose a task–centric programming model that enables decoupled, asynchronous execution. This model unlocks massive opportunities for parallelism, and allows storing task context information on-chip. To best utilize the proposed programming model, we design a domain-specific hardware accelerator using its data path and memory subsystem design to cater to the unique workload characteristics of temporal motif mining. To further improve performance, we propose a novel optimization called search index memoization that significantly reduces memory traffic. We comprehensively compare the performance of Mint with state-of-the-art temporal motif mining software frameworks (both approximate and exact) running on both CPU and GPU, and show 9×–2576× benefit in performance.

***Keywords*-hardware accelerator; programming model; temporal motif mining;**

I. Introduction

Graphs (or networks) provide a general and useful abstraction for modeling complex phenomena, *e.g.,* social and communication networks in computational social science, protein-protein interaction networks in biology, and transaction networks in finance [6], [35]. Small subgraph patterns, referred to as *motifs*, play a crucial role in understanding the structure and function of complex systems encoded as a graph [5], [48], [64]. Mining motifs is one of the central problems in network science [44].

Most real world phenomena are not static. Static graphs aggregate the interactions that occur over networks by omitting the temporal information. While analyzing static graphs is useful, doing so completely disregards the dynamics occurring over the graph. For example, in the case of an email exchange network, a static graph renders two users "connected" irrespective of the number of emails exchanged between them. This leads to severe information loss. Temporal graphs, on the other hand, retain this information by maintaining a list of all interactions and their respective timestamps. Therefore, temporal graphs capture richer information compared to static networks [30], [51].

Temporal motifs are one of the fundamental building blocks of temporal networks, analogous to how static motifs are for static networks. Temporal motifs have been shown to be useful in user behavior characterization on social and communication networks [30], [31], [33], [53], predicting peptide binding in structural biology [43], characterizing the structure and function of biological networks in bioinformatics [23], monitoring energy disaggregation on electrical grids [63], detecting fraud in financial transaction networks [19], and detecting insider threats in an organization's network [18], [38]. Furthermore, local motif counts have been shown to resolve symmetries and improve expressive power of graph neural networks [11]. Similarly, local temporal motif counts can be used as a subroutine for calculating node features in temporal graph learning [59].

Despite such wide utility of temporal motif mining, existing software frameworks offer sub-optimal CPU performance. This is because of the high computational complexity and irregular memory accesses of this workload. Temporal motif mining adds a time dimension to an already computationally and memory intensive static graph mining problem [10], [12], [13], [15], [57], [71], [78]. Furthermore, accesses to the graph structure and temporal edges incur irregular memory accesses that negatively impact the memory system's performance. While several acceleration techniques have been designed to speed up static graph processing [2], [4], [7], [9], [20], [45], [46], [49], [55], [62], [66], [67], [70], [77], streaming graph processing [8], [56], and static graph mining [10], [12], [13], [57], [71], [78], no prior work targets temporal motif mining. Moreover, temporal motif mining exhibits unique workload characteristics compared to previously studied graph problems as it processes temporal properties along with structural constraints (the main focus of prior works).

In this paper, we present Mint—a novel hardware accelerator architecture and accompanying programming

model for efficiently mining temporal motifs. To best address the challenges of efficiently executing this irregular algorithm, the design goals of Mint are three-fold: (1) realize a high degree of parallelism, (2) achieve high hardware utilization, and (3) improve memory system performance. To this end, we propose a task–centric programming model that enables asynchronous execution. The task is defined as a basic unit of computation for mining temporal motifs, *e.g.,* searching for a new edge to match. The asynchronous nature of this model unlocks a high degree of parallelism. Additionally, decoupled execution allows better hardware utilization.

Mint further proposes a new hardware accelerator to best utilize the proposed programming model. The hardware architecture is motif-agnostic, and can be programmed to mine any arbitrary motif. The key features of the proposed hardware design include a hardware 1) *task queue* that dispatches tasks to compute units, 2) on-chip *context memory* that stores the key task context information to identify and advance the progress of an in-flight task, and 3) unique distribution of work to different compute units that perform on-chip context updates and off-chip graph traversal to find a new edge mapping. To further enhance the performance of this architecture, we make a key observation that the amount of node neighborhood data used by the algorithm reduces with respect to time. Based on this observation, we propose a novel optimization of memoizing the search index that significantly reduces the memory traffic of Mint.

We comprehensively evaluate the performance of Mint using detailed RTL models of proposed hardware and a C++ based cycle-accurate simulator. We compare the performance of Mint with state-of-the-art software frameworks running on a high-end server-grade CPU and a GPU. Mint outperforms the CPU implementations of Mackey *et al.* [38] and Paranjape *et al.* [53] by 363.1× and 2575.9× respectively, a GPU version of Mackey *et al.* by 9.2×, and PRESTO [61] by 16.2×, on average. Similar to Mackey *et al.* [38] and Paranjape *et al.* [53], Mint runs an exact mining algorithm, whereas PRESTO is an *approximate* mining algorithm. Using 28 nm commercial technology library, we implement Mint to find that it consumes just 28.3 mm^2 silicon area and 5.1 W.

Mint is the **first work** that designs a domain-specific accelerator for mining temporal motifs. The key contributions of this work are as follows:

- *Task-centric programming model* that allows for massive parallelism opportunities.
- *Hardware accelerator* architecture that uses its data path and memory design to cater to the unique properties of temporal motif mining.
- *Search index memoization* optimization that significantly reduces memory traffic.
- *Mint*—the first end-to-end system design for accelerating temporal motif mining that significantly outperforms existing software baselines running on a CPU by one–to–three orders of magnitude.

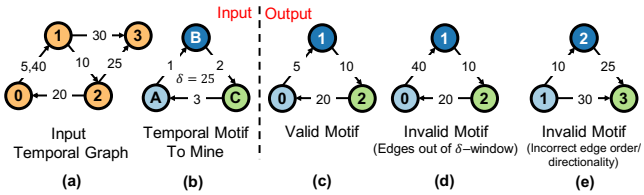


Figure 1. Example of $\delta-$temporal motif mining task. Depicted in (a) is the input graph, and (b) is the $\delta-$temporal motif. (c) presents a valid candidate for $\delta-$temporal motif in the input graph, (d, e) are invalid motifs due to violation of $\delta-$constraint and edge ordering, respectively.

II. Background

A. Problem Definition

A *temporal edge* is defined to be a timestamped directed edge between an ordered pair of nodes. We define a *temporal graph* as a collection of temporal edges. Formally, a temporal graph G is a collection of tuples $G = \{(u_i, v_i, t_i)\}_{i=1}^{m}$, where u_i and v_i are source and destination nodes of the edge (u_i, v_i), and $t_i \in \mathbb{R}^+$ is a timestamp of the edge. We assume that the timestamps of the edges in the temporal graph G are unique[1]. Strictly speaking, G is a *multi-digraph* as (a) the edges are directed, (b) there might be many edges between a pair of nodes, each one with a different timestamp; in this work, we refer to G as a graph for simplicity.

A *δ-temporal motif*[2] is defined as a sequence of l edges, $M = \{(u_i, v_i, t_i)\}_{i=1}^{l}$, that are time-ordered and occur within a δ duration, *i.e.,* $t_1 < t_2 < \ldots < t_l$ and $t_l - t_1 \leq \delta$. The problem of *temporal motif mining* is to mine occurrences of the δ-temporal motif M within a larger temporal graph G. In simple words, a δ-temporal motif is an occurrence of the sequence of edges in the graph G such that the first and last edges of this sequence occur *at most* δ time apart. It differs from the task of static motif mining in two ways: (1) in static motif mining, we are not interested in the *sequence* in which the motif's edges occur within G; (2) static motifs do not impose any constraints on the edge properties. Temporal motif mining may be interpreted as identifying subgraph isomorphisms with sequential and δ–constraints over edges.

As a simple example, consider mining of a three-cycle *δ-temporal motif* in an input temporal graph as shown in Fig. 1. Fig. 1(c) shows a valid motif because the edges between nodes 0, 1, and 2 in this motif follow edge ordering and occur within $\delta = 25$. On the other hand, Fig. 1(d,e) show invalid motifs either due to the δ-constraint violation or an incorrect edge order. In a static setting, however, all three motifs are valid as it does not account for edge ordering and timestamps.

[1]This assumption comes without loss of generality [38], [53].

[2]We sometimes refer to *δ-temporal motif* simply as *temporal motif* for brevity.

B. Real-world Applications

Temporal graphs capture a rich set of information compared to static graphs by storing dynamic interactions in addition to the graph structure. Mining temporal motifs has shown to be effective across several application domains including social and communication networks [30], [31], [33], [53], structural biology [43], bioinformatics [23], and finance [19].

In [31], the authors use temporal motifs as a tool to understand and quantify how information flows over a social network. Crucially, they demonstrate that it *cannot* be captured using its static counterparts. In financial transaction networks, certain types of temporal motifs can reveal artificial attempts to create high transaction volumes – an indication of potential financial fraud [19]. Features built with temporal motif distributions are shown to outperform their static counterparts in machine-learning based network classification [73]. In [38], authors show how temporal motif mining can be used to detect insider threats in an organization. *In summary, temporal motifs are one of the most fundamental properties (e.g., degree and centrality) computed over temporal graphs. As temporal graphs become ubiquitous, it becomes increasingly important to mine temporal motifs efficiently.*

C. Algorithmic Prior Work

Several algorithms have been proposed to mine motifs in temporal graphs. These algorithms can be broadly classified into two categories: (a) exact algorithms [32], [38], [53], and (b) approximate algorithms [37], [61], [74]. While exact algorithms aim to mine the precise temporal motifs in an entire input graph, approximate algorithms sample a subset of an input graph to estimate the number of matches in the entire graph. By limiting the amount of work, approximate algorithms achieve better scalability by reducing both computational and memory complexities. However, they suffer from inaccuracies in motif counts as they do not enumerate all motifs. In many scenarios, exact algorithms are still desired. For example, (1) in financial transaction networks, identification of *cycles*, a specific class of temporal motifs, indicates potentially fraudulent activity [19]; (2) in organization networks, certain temporal motifs can characterize insider threats [18], [38]. In such high-risk scenarios, it is crucial to employ exact mining algorithms to enumerate *all* instances of the desired motifs, instead of approximate mining. Furthermore, approximate algorithms use exact algorithms as subroutines to process a subset of nodes/edges [37], [61]. Therefore, Mint focuses on speeding up exact temporal motif mining; it is also *directly applicable* to accelerate approximate mining algorithms.

Exact algorithms can be further divided into two subcategories: (1) pattern-specific algorithms [32], [53], and (2) generic pattern-agnostic algorithms [38], [53]. While pattern-specific algorithms achieve better efficiency by using computation catered to a specific temporal motif, their applicability is limited. Furthermore, unlike static graph mining (*e.g.,* GraphPi [65], AutoMine [41], GraphZero [40]), no automatic framework exists for temporal motif mining that can discover optimized algorithmic schedules for arbitrary motifs. This requires designing hand-optimized algorithms for every new motif, which is error-prone and requires non-trivial programmer effort. Pattern-agnostic exact algorithms, on the other hand, can be used to mine any arbitrary temporal motifs. *In this paper, we focus on optimizing a state-of-the-art pattern-agnostic exact temporal motif mining algorithm proposed by Mackey et al. [38] that outperforms prior algorithmic works.*

D. Algorithmic Behavior

This section introduces the temporal motif mining algorithm proposed by Mackey *et al.* [38].

Data structures. The primary data structure used in this algorithm is a *temporal edge list*, stored in an array of structures. Each member of this array contains source and destination node IDs and a timestamp. Temporal edges in this array are sorted based on their timestamps. Additionally, the graph structure is stored in a compressed format to simplify retrieving incoming and outgoing edges from each node. This structure stores *indices* of temporal edges in the temporal edge list (instead of storing the temporal edges [69]). In addition to the graph structure, this algorithm stores key book-keeping information. This includes mappings between motif and graph nodes ($m2gMap[]$, $g2mMap[]$). A stack ($eStack$) of mined edge indices is used for Depth-First Search (DFS) traversal.

Algorithm. Mackey *et al.* [38] present a pattern-agnostic temporal motif mining algorithm that uses search tree exploration. Each node of the search tree matches an edge in a motif to an edge in the graph. Starting from the first edge (root node of the search tree), the algorithm iterates over edges of the temporal motif in a chronological order to find one match at a time, following a DFS tree traversal. Upon matching each edge, book-keeping information is updated.

Algorithm 1 presents this in detail. The outer while loop iterates over edges in an input motif. For each edge in a motif, the FINDNEXTMATCHINGEDGE() function tries to match a corresponding edge in a graph. By $\mathcal{N}_{out}(u)$ and $\mathcal{N}_{in}(u)$, we denote the list of outgoing and incoming edges of a node u, respectively. If a valid match is found, book-keeping structures are updated. Otherwise, a backtracking procedure voids previous matches using a stack following a DFS traversal order, and this process is repeated until all motifs are found. This algorithm performs most of its work in finding an edge to map. As shown in the algorithm, this procedure also takes into account whether or not either source and/or destination node of the motif edge have been mapped, and edge orderings to reduce the search space.

Algorithm 1 Generic temporal motif mining algorithm [38]

```
1: procedure TemporalMotifMining(G, M, δ)
2:     Input: temporal graph (V_G, E_G), motif (V_M, E_M), time limit δ.
3:     Output: temporal motifs
4:   // Initialize data structures: edge mapping, counters
5:   Initialize: m2gMap[u] = −1 ∀u ∈ V_M; g2mMap[u] = −1 ∀u ∈ V_G
6:               eCount[u] = 0 ∀u ∈ V_G, eStack = [], e_M = −1, e_G = −1, t' ← ∞
7:   while true do                                   ▷ Loop until all motifs found
8:     e_G = FindNextMatchingEdge()  ▷ Search: find a graph edge to match
9:     if e_G is valid then
10:      UpdateDataStructures()      ▷ Book-keeping: update data struct
11:    e_G += 1
12:    while e_G > |E_G| or time(e_G) > t' do ▷ Backtrack: void previous mapping
13:      if eStack is not empty then
14:        e_G = eStack.pop() + 1
15:        if eStack is empty then t' ← ∞
16:        eCount[u_G] −= 1, eCount[v_G] −= 1   ▷ Reduce mapped edge cnt
17:        if eCount[u_G] == 0 then             ▷ No edges of u_G mapped
18:          u_M ← g2hMap[u_G]
19:          g2hMap[u_G] = −1, h2gMap[u_M] = −1          ▷ Free u_G, u_M
20:        if eCount[v_G] == 0 then             ▷ No edges of v_G mapped
21:          v_M ← g2hMap[v_G]
22:          g2hMap[v_G] = −1, h2gMap[v_M] = −1          ▷ Free v_G, v_M
23:      else
24:        return results
25:
26: procedure FindNextMatchingEdge( )                ▷ Find a new mapping
27:   (u_M, v_M) = E_M[e_M]
28:   (u_G, v_G) = m2gMap[u_M], m2gMap[v_M]
29:   // Gather candidate edges to match with the next motif edge
30:   if u_G ≥ 0 and v_G ≥ 0 then          ▷ Both u_G, v_G mapped to motif nodes
31:     S ← {e ∈ N_out(u_G) / N_in(v_G) : t_e > time(e_G)}  ▷ Irregular access + filter
32:   else if u_G > 0 then                 ▷ Only u_G mapped to a motif node
33:     S ← {e ∈ N_out(u_G) : t_e > time(e_G)}              ▷ Irregular access + filter
34:   else if v_G > 0 then                 ▷ Only v_G mapped to a motif node
35:     S ← {e ∈ N_in(v_G) : t_e > time(e_G)}               ▷ Irregular access + filter
36:   else                                 ▷ Both u_G, v_G not mapped
37:     S ← {e ∈ E_G : t_e > time(e_G)}     ▷ Search space is an entire edge list
38:   // Return the first valid candidate edge that satisfies temporal constraints
39:   for each edge e in S do
40:     if e is not mapped and time(e) < t' then
41:       return e
42:
43: procedure UpdateDataStructures( )                ▷ Add a new mapping
44:   if e_M == |E_M| − 1 then                          ▷ Entire motif found
45:     Create a motif H from edges in eStack; add H to results.
46:   else                                              ▷ Partial motif found
47:     (u_G, v_G) ← E_G[e_G], (u_M, v_M) ← E_M[e_M]
48:     m2gMap[u_M] = u_G, m2gMap[v_M] = v_G  ▷ Map motif node to graph node
49:     g2mMap[u_G] = u_M, g2mMap[v_G] = v_M  ▷ Map graph node to motif node
50:     eCount[u_G] += 1, eCount[v_G] += 1     ▷ Increment mapped edge cnt
51:     if eStack is empty then                ▷ e_G is the first matched edge
52:       t' ← time(e_G) + δ                   ▷ Upper bound on the motif's end time
53:     eStack.push(e_G); e_M += 1
```

III. Why Design A New Accelerator?

This section argues the need of designing a new accelerator for accelerating the temporal motif mining problem.

A. Essence of Optimizing This Workload

Wide applicability. Static graphs do not capture rich dynamics that occur over networks [30]. Temporal networks are ubiquitous in domains ranging from communications, biological sciences, and finance. Temporal motifs are *fundamental* building blocks that constitute a temporal network [30]. Therefore, counting and mining temporal motifs is one of the primary tasks in temporal network analysis [22]. As we discussed in §II-B, temporal motif mining is widely applicable across several critical application domains. These include user characterization in social and communication networks [30], [31], [33], [53], understanding and explaining biological phenomena [43], monitoring electrical power grids [32], and machine learning on temporal graphs [11], [59], [73]. With the explosion of online content and digital footprint generated by social networks, temporal graphs are getting larger and richer by the day. *Therefore, designing optimization techniques for such widely applicable primitive operations is highly desirable.*

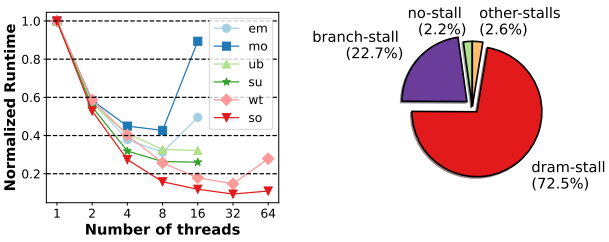


Figure 2. Performance scaling of M1 mining on different datasets (left), and CPU stall distribution (right) for mining M1 on a representative wiki-talk dataset.

Algorithmic complexity. Let $|E_G|$ and $|E_M|$ denote the number of edges in the graph and the motif, respectively, and k denote the expected number of edges occurring in G within the duration δ. The worst-case algorithmic complexity of Algorithm 1 is $\mathcal{O}\left(|E_G| \cdot k^{|E_M|-1}\right)$: it scales linearly with $|E_G|$, polynomially with k, and *exponentially* with $|E_M|$. Intuitively, (i) increasing the temporal limit δ of a motif increases the *width* (*i.e.,* the number of nodes to visit in E_G for each edge in E_M) of the DFS search tree; and (ii) increasing motif size $|E_M|$ increases the *depth* of the DFS search tree. Therefore, the resulting complexity increases. For example, mining a 5-edge motif with temporal limit δ of 1 hour on the stackoverflow dataset implies $|E_G| = 32M, |E_M| = 5$, and $k_{avg} \approx 1500$, *leading to unreasonably high complexity*.

B. Workload Characterization and Optimization Opportunities

To understand the limitations of running existing temporal motif mining software on a commercial CPU, Fig. 2 shows its performance scaling and stall distribution. The figure shows the runtime of mining M1 on different datasets, normalized to a single–thread performance. The figure shows that the performance scaling saturates beyond 8–32 threads. For small datasets, the performance degrades by adding more threads as threading overhead dominates the execution time.

To better understand this trend, Fig. 2 (right) shows the stall distribution for mining M1 on a representative wiki-talk dataset, based on the CPI stack methodology [17]. For this experiment, we use a 32-thread configuration with three levels of cache hierarchy, 2 MB LLC slice/core. This distribution shows that the CPU spends 72.5% and 22.7% of the execution time stalled on DRAM and branch mispredictions. The DRAM stall is due to two reasons: irregular memory accesses to access graph structure and

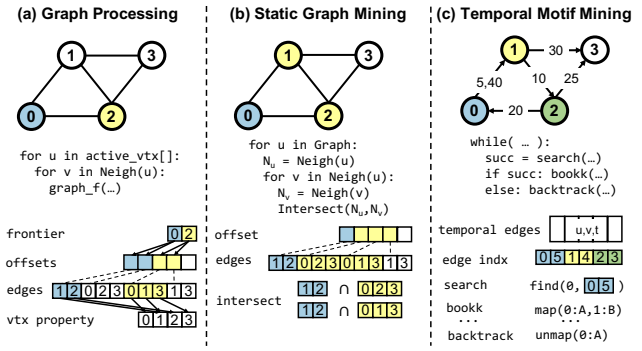


Figure 3. Unique workload characteristics in terms of data structures and algorithms employed in (a) graph processing, (b) static graph mining, and (c) temporal motif mining.

neighborhood filter operations that waste memory bandwidth (shown in lines 31, 33, 35 in Algorithm 1). Furthermore, data-dependent control flow instructions in lines 30–36 and lines 13–20 cause frequent branch mispredictions. *This calls for developing new acceleration techniques that can alleviate the memory and control-flow bottlenecks of this workload.*

C. Unique Workload Characteristics

At an algorithmic level, most prior works optimizing graph algorithms [2], [3], [4], [10], [12], [13], [15], [20], [45], [49], [55], [57], [67], [71], [77], [78] work on static graphs, whereas temporal motif mining operates on a temporal graph that adds a time dimension to the problem. While streaming graph processing accelerators [8], [56] also operate on dynamic graphs, they mostly optimize traditional graph computations (*e.g.,* PageRank). The reason is that streaming graphs measure *properties of the accumulated static graph over time*; this is *different* from temporal graph analysis (our setting), where the goal is to analyze *temporal properties* of a graph [60]. The edges of a temporal graph carry time information; therefore, the edges are *ordered*. In fact, this notion of order is *central* to temporal graph analysis. In streaming graphs, on the other hand, although the edges arrive at different points in time, they are treated as "updates" performed to the underlying accumulated static graph. Here, in stark contrast to temporal graph analysis, the graph properties remain unchanged even if the order of edges is permuted.

To motivate the necessity of designing a new accelerator for temporal motif mining, this subsection points out differences between its workload characteristics and other well-studied workloads, *i.e.,* graph processing and static graph mining. Fig. 3 shows the difference between generic pseudo-code, data structures, and computation patterns used in three algorithm categories. Graph processing algorithms (*e.g.,* PageRank and SSSP) typically pick a vertex from an active list, access its neighbors using offset and edge lists, and updates vertex properties as shown in Fig. 3(a). This incurs irregular data-dependent access into the offset, edge, and property lists. Graph mining algorithms, on the other hand, typically iterate over all vertices, find vertex neighborhoods, and compute set operations (*e.g.,* intersection) to mine subgraphs. The unique feature of this algorithm is the set operation computation, not present in graph processing algorithms. This has motivated the designs of novel architectures [10], [12], [13], [15], [57], [71], [78] for accelerating static graph mining.

As discussed in §II-D, temporal motif counting uses temporal edge list, instead of a static edge list. The key difference between these data structures is that the edges in a temporal edge list are sorted by their timestamps. Therefore, unlike the edge list used in static algorithms, outgoing/incoming edges from the same node are not stored contiguously for temporal motif mining. Due to this design, the edge list for temporal algorithm stores edge indices instead of neighbor IDs. Furthermore, mining temporal motifs performs search, book-keeping, and back-tracking (Algorithm 1), where it spends a majority of execution time fetching neighborhood and searching for the first edge with a timestamp larger than the previously matched edge (lines 31, 33, 35 in Algorithm 1). Unlike static subgraph mining, temporal motif mining does not employ set operations as primitive computational blocks. Moreover, the amount of work performed by static/temporal motif algorithm is roughly proportional to the number of matched motifs, because each match requires a full expansion of the search tree. As shown by prior work [38], the ratio of number of matched static to temporal motifs vary by orders of magnitude. Depending on the input graph and motif, this ratio can be significantly higher or lower than 1. Therefore, the amount of work in static and temporal motif mining algorithms can be vastly different. *Due to the unique layout of data structures and computation patterns in temporal motif mining that lead to significantly different amounts of algorithmic work, it cannot be readily accelerated using prior techniques, which calls for designing a new accelerator for this problem.*

IV. Task–Centric Programming Model

Motivated by the workload characteristics of temporal motif mining (§III-B), this section presents a novel task–centric programming model. The goals of this model are two-fold: (a) enable asynchronous execution to unlock massive parallelism and improve hardware utilization, and (b) reduce off-chip memory traffic.

A. Task: A Unit of Computation

A task is referred to as a single unit of computation used in temporal motif mining. Algorithm 1 performs three unique types of computations: 1) **search**: find the next edge to map, 2) **book-keeping**: update key metadata information when a valid edge is found, and 3) **backtrack**: void the last mapped edge in metadata structures if no valid match is found. To

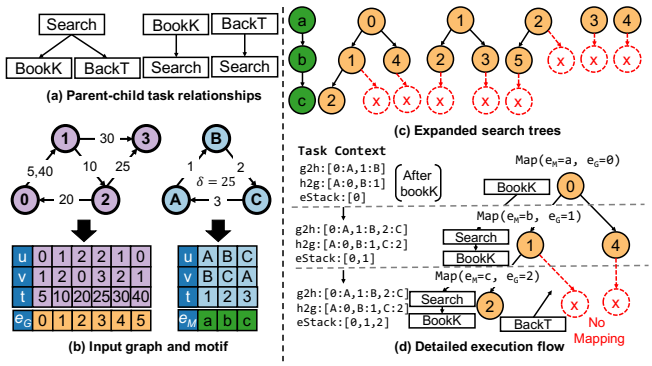


Figure 4. (a) Parent-child relationship between different task types, (b) an example input graph, temporal motif, and their temporal edge lists, (c) expanded search trees to mine a temporal motif, (e) a walk-through example of proposed programming model with task and metadata progression.

capture this algorithmic behavior, we propose to represent these three functions as *tasks*.

Temporal motifs are mined using search tree expansion. Tasks that initiate the mining of a motif (that we call *root tasks*) are generated by matching the first motif edge with different temporal graph edges in chronological order. To expand the search trees further, subsequent tasks are generated by their parent tasks based on the outcome computation. For example, a search task would generate either a book-keeping or a backtracking task based on whether a matching edge is found or not. Fig. 4(a) shows the parent-child relationship between different task types. This allows a natural, incremental flow of task information from parents to children. A task terminates upon its completion. Upon termination, a child task is spawned if additional work is needed to traverse the search tree. The task generation stops once the whole search tree has been explored, and a new root task is generated.

B. Task Context

Each parent task communicates its progress to its child tasks to continue the mining process. We propose to capture this information in terms of a *task context*. A task context includes (a) task type, (b) last matched motif edge index (e_M), (c) last matched input edge index (e_G), (d) a mapping of graph nodes to motif nodes ($g2mMap[]$) and vice versa ($h2gMap[]$), (e) a stack of mapped temporal edge indices (*eStack*), and (f) initial timestamp ($firstEdgeTime$). A context stores minimal information required to traverse the search tree. *Therefore, a task context enables execution decoupling between the parent and child tasks.*

Memory requirement. The memory requirement of a task context is low. The task type, edge IDs, and temporal information are all integers, and can be stored with $\mathcal{O}(1)$ memory complexity. The node maps and the edge stack, on the other hand, grow linearly with the number of edges in input temporal motifs (*i.e.,* memory complexity $\mathcal{O}(|E_M|)$. As shown by prior algorithmic works [32], [37], [38], [53], [61], [74], a practical temporal motif size in real-world applications is up to eight edges. Using this conservative estimation, the memory requirement of a task context is 178 B. *This negligible memory requirement allows several task contexts to be stored on-chip and accessed at low latency in an accelerator.*

C. A Walk–Through Example

Fig. 4(b-d) demonstrate a walk-through example of the proposed programming model. Fig. 4(b) shows an example input graph, temporal motif, and their temporal edge lists. Fig. 4(c) shows the expanded search trees to mine the input motif. Note that each node of these trees maps one edge in the graph to a motif edge.

Fig. 4(d) expands the left-most search tree to explain how the programming model works. As discussed in §IV-A, the root task automatically maps edges in the graph to the first motif edge in chronological order. Therefore, the first task performs book-keeping to map $e_M = a$ to $e_G = 0$. As shown in the simplified task context, graph nodes *0* and *1* are mapped to motif nodes *A* and *B* using $g2hNodes[]$ and $h2gNodes[]$. The matched graph edge $e_G = 0$ is pushed to the stack. Additionally, e_G, e_M, and $firstEdgeTime$ are also updated (not shown due to limited space).

This book-keeping task spawns a search task that finds the next edge to map to $e_M = b$. Using the graph structure and temporal information, this step finds $e_G = 1$ and spawns a book-keeping task to update the task context. This book-keeping task extends the context by mapping graph nodes *2* to motif node *C* as well as pushing $e_G = 1$ to the stack. This process continues until either a full motif is mined or if the search task cannot find any edge to map. Fig. 4(c) shows that there is no dependency between traversing different search trees, which results in traversing the different search trees in parallel *asynchronously*. Furthermore, in the proposed programming model, a task context is the only information necessary to advance the search. This naturally allows asynchronous, parallel task execution. *In sum, the proposed programming model with the right hardware design can achieve high throughput.*

D. Code Transformation

Fig. 5 shows the conversion of temporal motif mining code from an edge-centric to task–centric programming model. To achieve this, a programmer has to define the `TaskContextType` class (Fig. 5(a)). This includes memory allocation for context information, and helper functions to update the context. Additionally, the programmer needs to convert the main procedure used in the core algorithm. Lines 5–12 in Fig. 5(b) show these changes. The main data structure is the task queue. Because search trees can be traversed in parallel, several worker threads can dequeue pending tasks from, and enqueue new tasks back to the

```
class TaskContextType {

  public:
    // define helper functions

  private:
    bool _busy = false;
    TaskType _type;
    int _eG = -1, _eM = -1;
    MapType _h2gMap, _g2hMap;
    StackType _eStack;
    int _firstEdgeTime = -1;

};
```

(a) Task context class

```
int TemporalMotifMining(…) {
  int num_matches = 0;
  TaskQuqueType t_queue;

  while(true) {  // parallelize search trees
    if(t_queue.empty() && eG==n_graph_edges())
      break;
    TaskType task = t_queue.dequeue();
    task.process(&num_matches); // SR, BK, BT
    t_queue.enqueue(task.child_task());
      // SR->BK/BT, BK/BT->SR
  }
  return num_matches;
}
```

(b) Task-centric temporal motif mining

Figure 5. Task–centric temporal motif mining.

queue for several tree expansions. The `task.process()` function executes one of three tasks: search, book-keeping, and backtracking. If a leaf node of the search tree successfully finds a match, `num_matches` is incremented. As presented in §IV-C, traversing different search trees are independent of each other. Because this is a generic algorithm that can be used to mine any arbitrary temporal motif in any input temporal graph, the programmer effort modifying this code can be easily amortized over several executions.

V. Accelerator Architecture

To best utilize the proposed programming model, this section proposes a novel hardware accelerator architecture.

A. Design Overview

Fig. 6 shows the hardware accelerator design of Mint. It contains four memory structures, *i.e.,* task queue, target motif, context memory, and on-chip caches, and two computation blocks, *i.e.,* context manager and search unit. The task queue is a hardware FIFO queue that queues root tasks (§IV-A). This root task is offloaded directly to the context manager with a book-keeping task to initialize the search tree expansion. The target motif, programmed by the host CPU, stores the motif being mined, making Mint a *generic and motif-agnostic* temporal motif mining hardware design.

The context memory stores metadata information to keep track of task progress. This memory is updated by the context manager during the book-keeping/backtracking phase. The search unit, on the other hand, reads from this context memory to mine a graph edge. Finally, the on-chip caches are used to cache the graph structure and temporal edge list data to reduce the memory latency of the search unit. The on-chip context manager only updates the contexts of in-flight tasks, and does not communicate with DRAM.

As detailed in §IV, a task can have three types: search, book-keeping, and backtracking. The context manager executes book-keeping and backtrack tasks. The search unit, on the other hand, is solely responsible for the search task. The search unit is further divided into blocks: the (1) dispatcher, and (2) search engine. The dispatcher reads an updated context, and dispatches work to the search engine. The search engine, in turn, consumes this task, and mines a graph edge to match a temporal motif edge. Upon completion of a search task, the search engine offloads either a book-keeping or backtracking task to the context manager for updating the task context, depending on the success or failure of the search.

The context manager only performs on-chip accesses to update context memory. These accesses take a single cycle. On the other hand, the search engine fetches data from DRAM, which takes multiple cycles. While some part of this latency is reduced by on-chip caches, multiple search units are necessary to exploit memory-level parallelism. Therefore, Mint employs several search engines that work on independent search tasks in parallel. To match the throughput of search engines, several context managers and memory instances are also used. While it is possible to use a fully asynchronous programming model, where any compute engine can pick up any pending task from the queue, this requires costly on-chip crossbars and routing logic, and a multi-ported task queue to enable an architecture with a large number of compute units. To simplify this microarchitectural design and routing logic, Mint's context manager, context memory, dispatcher, and search engine work in tandem to traverse a search tree. While this architecture also allows an asynchronous task execution model, limiting the location of task offload greatly simplifies the design parameters and saves silicon area/power by avoiding costly routing logic. This, however, does not sacrifice performance because each context memory instance is busy when the assigned search engine mines an edge. This claim is further verified by the high bandwidth utilization of Mint (§VIII), showing that a fully flexible all–to–all connection between context managers and search engines is unnecessary.

B. Hardware Component Design Details

Target motif memory. This is a small register file that holds the target motif. For each temporal motif edge, it stores the source and destination IDs, and one delta time for an entire motif. Because the motif only has an edge ordering, a simple register file design is sufficient, where it is possible to use the chronological edge number e_M as an index. Prior works mine motifs with up to eight edges. Therefore, Mint supports temporal motifs of up to eight edges.

Task queue. The task queue is used to store and offload root book-keeping tasks. Fig. 6(b) shows the fields in each entry. Each queue entry stores a root task packet that contains a book-keeping task with additional information about mapping the first motif edge M_{edge} with different graph edges G_{edge} in chronological order as shown in §IV-A. Therefore, each task queue entry stores the graph edge index e_G. Using e_G, Mint compute units can obtain source/destination graph nodes and edge timestamps from DRAM. Task queue initiates a search tree traversal by offloading a book-keeping task to the context manager. After this, a context manager works

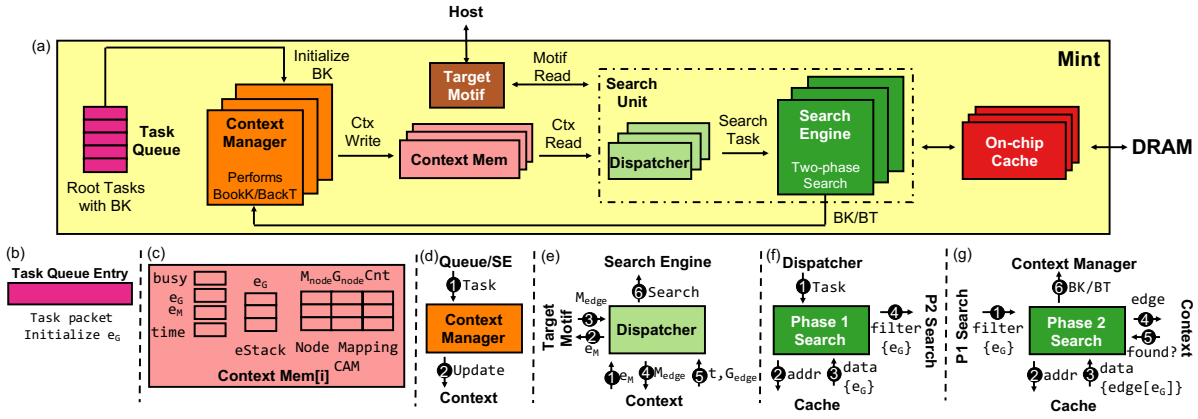


Figure 6. Hardware design overview of Mint: (a) overall architecture, (b) task queue entry, (b) an instance of context memory, and workflows of (d) context manager, (e) search unit dispatcher, (f) first phase of search engine, and (g) second phase of search engine. Parts (b-g) show Mint hardware components and their interactions with the rest of the system.

with a search unit to expand the rest of the search tree without communicating back to the queue.

Context memory. Fig.6(c) shows the context memory design, where each instance stores task context information. This includes a set of registers, a stack, and a Content Addressable Memory (CAM). Context registers store the task status (*i.e.,* busy or available), indices of the last mapped edge (e_M and e_G), and the timestamp of the last mapped graph edge. The stack *eStack* stores indices of previously mapped edges. The stack is used by the context manager for backtracking. The CAM memory stores the node mapping information, which mimics $g2mMap[]$ and $h2gMap[]$ in hardware. The design decision of using a CAM is to quickly search which motif node is mapped to which graph node, and vice versa. Additionally, the CAM also stores the number of times a graph node is mapped (*eCount* in Algorithm 1).

On-chip cache. This is a standard multi-bank, multi-port set associative SRAM cache that reduces the latency of search by caching the graph structure and temporal information.

Context manager. The context manager updates the context memory while performing book-keeping or backtracking. As shown in Fig. 6(d), it accepts task packets either from the queue (only the root task) or the search engine (SE) (❶), and updates specific parts of context memory based on the task type (❷). For book-keeping, the manager expands a context for a newly matched edge that includes pushing an edge index to *eStack*, expanding the node mapping CAM, and incrementing their connection counts. It also updates e_G, e_M, and *time* registers to reflect the state of the most recent search. For backtracking, a context manager pops an entry from the stack, voids node mappings, and updates edge index and time registers to invalidate the last edge mapping.

Search unit dispatcher. After the context memory is updated, the dispatcher reads this context and offloads a search task to the search engine. As shown in Fig. 6(e), the dispatcher first reads an updated motif edge index (e_M), and reads edge information from the target motif (❷-❸). Using this information, the dispatcher reads context memory (❹-❺) and finds the timestamp of the last mapped graph edge, and node IDs in the graph that are mapped to source and/or destination node IDs of the temporal edge that the search engine will mine next. Using this information, the dispatcher offloads a search task packet to a search engine (❻).

Search engine. A search engine performs a two-phase search in an attempt to match a motif edge to a graph edge. The first phase finds a set of graph edge indices that might map to a motif edge, and the second phase finds an exact edge. Search phase 1 (Fig. 6(f)) accepts a search task from the dispatcher (❶) that contains source/destination IDs of the motif edge being mined and previously mapped graph node IDs (if any). Using these graph node IDs, the search engine fetches its outgoing/incoming edge indices (❷-❸) (similar to lines 30–37 in Algorithm 1). Additionally, it filters edge indices to find all edges with timestamps after the last mapped graph edge. This is simply done by finding a subset of neighbor edge indices greater than e_G read from the context memory. In contrast to software that employs binary search, Mint employs linear search as it is possible to efficiently perform this operation in hardware by streaming edge index cache lines using a series of comparators in parallel. These filtered edge indices are then processed in the second search phase (❹).

Fig. 6(g) shows the second search phase that finds an exact edge to map. Using filtered edge indices from phase 1 (❶), this phase first fetches temporal edges from memory (❷–❸). These edges are examined for both structural and temporal constraints by reading the context information (❹) to find the first valid edge that matches a motif edge. Resolving structural constraints includes ensuring that either the new

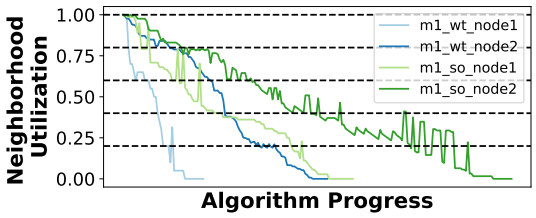


Figure 7. Reduction in the neighborhood data utilization for two sampled nodes while mining M1 on wiki-talk and stackoverflow. The x-axis represents the progress of an algorithm.

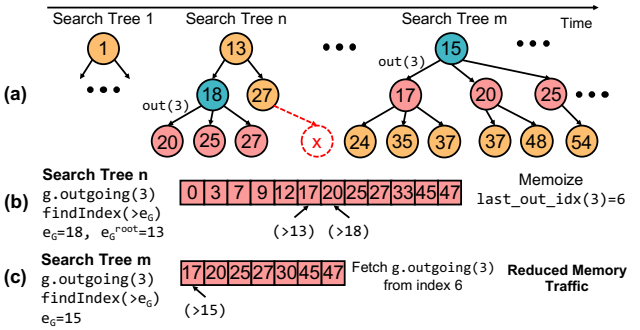


Figure 8. Design optimization of search index memoization to reduce memory traffic. (a) Progression of the algorithm with respect to time, (b) expanding the blue node in tree n by searching current $e_G = 18$ and $e_G^{root} = 13$, and (c) reduced search operation computation while expanding the blue node in three m due to memoization.

graph nodes are not mapped earlier, or mapped to the same nodes in the motif that we are trying to match. Resolving temporal constraints includes checking edge timestamps against max edge time for the motif being mined, to verify the delta-time requirement. If either of these constraints are violated, a graph edge is discarded and the search engine examines the next graph edge. Based on whether a valid edge is found or not (❺), the search engine offloads either a book-keeping or a backtracking task back to the context manager (❻).

Crossbar. In this architecture, there is only one crossbar that resides between the task queue and all context managers. Because this is a one–to–all connection, there is no arbitration needed that further reduces area and power. Each search engine only serves its paired context memory because a context memory will not generate a new search task until the search engine returns the result of the previous search tree expansion. Therefore all connections within a set of context memory, search engine, dispatcher, and context manager are local, eliminating the need for convoluted NoCs or crossbars.

VI. Design Optimizations

This section discusses a novel design optimization to reduce the memory requirement of phase 1 search. Additionally, we briefly discuss two standard optimizations that we tried that did not result in fruitful performance improvement.

A. Search Index Memoization

The goal of this optimization is to reduce the memory traffic in search phase 1 by fetching mostly useful data. As detailed in §V-B, the first phase of search fetches outgoing/incoming neighbors of a node, and filters them based on the current e_G. Lines 31, 33, 35 shows this filter operation in Algorithm 1. To better understand the behavior of this operation, Fig. 7 shows the utilization of neighborhood data with respect to time. Intuitively, due to chronological order of mining edges, as the algorithm progresses, the resulting filtered edges have higher timestamps. Because node neighborhoods store edge indices in increasing time fashion, the neighborhood utilization decreases with respect to time. Notably, this is not a problem in the software implementation [38] as it employs binary search. *This results in wasted DRAM bandwidth and on-chip cache resources.*

To prevent futile data fetches, we propose a novel optimization to memoize the search result. For each node, we memoize the resulting index of the last search. Because search is performed in a chronological order, it is guaranteed that the edges discarded in any search operation will never be used in its subsequent search operations. We use two data structures for memoizing the previous search result for incoming and outgoing neighborhoods. Because the amount of memoization memory grows linearly with the number of graph nodes, Mint stores these data structures in DRAM.

Fig. 8 presents this optimization with an example. Suppose that an outgoing neighborhood of node 3 is accessed to expand the blue tree nodes ($e_G = 15, 18$). The outgoing neighborhood of node 3 contains 12 edges with indices $0-47$ (shown in pink array). The first time that the search tree labeled n accesses outgoing neighbors of node 3, it fetches the entire neighborhood and searches for data elements greater than $e_G = 18$. The proposed optimization remembers the index of the first edge that occurs after the $e_G = 13$ of the root node ($last\_edge\_idx(3) = 6$). The next time that search tree m accesses outgoing neighbors of node 3, it only fetches all neighbors after index 6. This results in saving 5 unnecessary data fetches, and overall reduction in memory traffic.

The reason behind using an e_G of the root node for memoization is that all edges searched in any tree is guaranteed to have higher timestamps compared to the root nodes' edges from previous trees. However, as we expand search trees, there is no relation between the edge timestamps of non-root nodes in different trees. For example, Fig. 8 shows that outgoing neighborhood of node 3 is accessed by $e_G = 18$ in the earlier tree and $e_G = 15$ in the latter tree. Therefore, memoizing the index of edges greater than $e_G = 18$ for search tree n would result in incorrect result as it would miss an edge index 17 while expanding the search tree m.

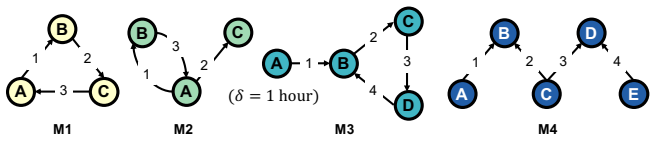


Figure 9. Temporal motifs used for evaluation.

Graph	#Vertices	#Temporal Edges	Size (MB)	Time span (days)
email-eu (em)	986	332.3k	7.6	808
mathoverflow (mo)	24.8k	506.5k	12.0	2350
ask-ubuntu (ub)	159.3k	964.4k	24.5	2613
superuser (su)	194.1k	1.4M	36.0	2773
wiki-talk (wt)	1.1M	7.8M	196.7	2320
stackoverflow (so)	2.6M	36.2M	1493.0	2774

Table I
TEMPORAL GRAPH DATA SETS USED FOR EVALUATION.

B. What Didn't Work?

In addition to this novel optimization, we try two other standard optimizations employed by prior graph accelerators: (a) task coalescing, and (b) prefetching. These optimizations, however, did not yield reasonable performance improvements. First, task coalescing attempts to reduce the amount of memory traffic by coalescing phase 1 search tasks that access the same node neighborhoods. While in theory it reduces the number of memory accesses, its performance is very close to a non-task coalescing baseline because the cache lines only need to access DRAM once, and subsequent tasks can access this data from cache. Second, we attempted neighborhood prefetching for phase 1 and phase 2 of search. However, a detailed microarchitectural analysis of Mint shows that search engines are waiting for DRAM accesses for more than 98% of time and utilize more than 60% of peak DRAM bandwidth. Adding prefetching hurts performance because of high bandwidth demand and cache pollution. Therefore, Mint does not implement these optimizations.

VII. EVALUATION METHODOLOGY

A. Algorithms and Datasets

Algorithms. As discussed in §II-C, we use a generic, exact temporal motif mining algorithm for our study. Similar to prior works [38], [53], we mine four unique motifs (**M1-M4**) from three to five nodes in size (Fig. 9) with δ=1 hour. Due to long simulation times and limited space, we limit our evaluation to these motifs. However, Mint is a generic accelerator, and can be used out-of-the-box to mine any motif.

Datasets. Similar to prior works [38], [53], we use six real-world temporal graph datasets for evaluation as shown in Table I selected from SNAP [34]. These datasets are diverse in terms of their sizes from small (email-eu) to large (stackoverflow), and connectivity. email-eu is an email-exchange network between users at a large European research institute. mathoverflow, ask-ubuntu, superuser, and stackoverflow are interaction networks between users on Math Overflow, Ask Ubuntu, Super User, and Stack Overflow, in terms of comments, questions, and answers. wiki-talk is a user-editing network of pages on Wikipedia.

Component	Modeled Parameters
Context Manager	512× context manager instances that updates context memory
Search Unit	512× dispatchers, 512× two-phase search engines
Task Queue	1× queue, 16-entry, 4 B memory per entry, 1 cycle task dequeue latency
Context Memory	512× context instances, each instance has metadata registers, edge stack, and node connectivity CAM, 2 cycle access latency
On-chip Cache	64× cache banks of 64 KB SRAM cache (4 MB total), 4-way set associative, 2 cache ports per bank, 64 B block size, 32 MSHR per bank, 2 cycle access latency
DRAM	8-channel DDR4-3200, 204.8 GB/s peak bandwidth

Table II
MINT SYSTEM CONFIGURATION.

B. Baseline Hardware Platforms

To run the software baselines, we use a dual-socket server with two AMD EPYC 7742 processors, each with 64 physical cores (128 SMT threads). The aggregate Last Level Cache (LLC) size on each CPU is 256MB. The main memory in the system is 8-channel DDR4-3200 with a 1.5TB capacity. As shown in §III-B, the performance of temporal motif mining does not scale linearly with the number of threads. For each workload, we sweep the number of threads from 1 to 256, and choose the best-performing configuration for comparison. In addition to the CPU baseline, we compare the performance of Mint with an NVIDIA GeForce RTX 2080 Ti GPU.

C. Simulation Infrastructure

Table II shows the system configuration of Mint. It employs one task queue, and 512 context managers, search engines and context memory instances as detailed in §V-A. We use a 64-bank 64 KB on-chip SRAM cache (4 MB total), and 8-channel DDR4-3200 DRAM (same as CPU baseline).

To accurately estimate the performance of Mint, we implement a detailed two-phase simulation methodology. First, we model all hardware components (except caches) using System Verilog HDL. We synthesize this design using a commercial 28 nm technology library using the Synopsys Design Compiler. We use Synopsys PrimeTime for vector-based power estimation. Using detailed post-synthesis RTL simulations, we extract the critical path delay of our circuits and set Mint clock frequency at 1.6 GHz. Additionally, we collect the power and area numbers using RTL. We use CACTI [47] to estimate the performance/power/area of SRAM–based caches.

Second, to estimate end-to-end performance, we implement a cycle-accurate C++ simulator. This simulator faithfully models all system components, their RTL–based latencies, and their interactions. Several microarchitectural events are modeled in detail, including task queue dequeue, and stalls due to cache port contention, structural hazards at search engine, Miss Status Handling Registers (MSHRs) of cache,

and memory controller. To model DRAM, we use Ramulator [29]. We verify simulator functionality by matching its compute and memory traces with an instrumented software baseline ensuring no events are missed.

D. State-of-the-art Baselines

Mackey *et al.* [38] CPU. This is a state-of-the-art generic temporal motif mining algorithm uses a DFS-based search tree traversal. We convert their code into a task–centric multi-threaded implementation (similar to proposed programming model) using work stealing OpenMP threads.

Mackey *et al.* [38] CPU w/ Memoization. This baseline implements our proposed search index memoization optimization in software on a Mackey *et al.* CPU baseline. Memoized indices are stored in a dedicated array in main memory. Because the indices are memoized based on the e_G of a root node (§VI-A), two search operations are triggered—one to find the memoization index, and the other to find e_G of the current node. All search operations use binary search.

Paranjape *et al.* [53]. This is an exact mining algorithm that first mines static subgraphs, and then resolves temporal constraints using a dynamic programming problem.

PRESTO [61]. PRESTO proposes a scalable edge sampling technique for approximate mining. It uses Mackey *et al.* [38]'s algorithm to mine motifs on a subset of edges.

Mackey *et al.* [38] GPU. This is a CUDA implementation of a state-of-the-art generic temporal motif mining algorithm running on an NVIDIA GPU. This baseline uses an in-house implementation as no open-source implementation of Mackey *et al.*'s algorithm exists.

Static graph mining accelerator FlexMiner [13]. Although FlexMiner does not support temporal motif mining, we divide this workload into two phases similar to a baseline algorithm presented in Paranjape *et al.* [53]: (1) mine static subgraphs by ignoring temporal information, and (2) use results of the first phase to mine temporal motifs by resolving temporal constraints. We use a state-of-the-art graph mining framework GraphPi [65] to find the performance of phase 1 on a CPU baseline (§VII-B). To find FlexMiner performance, we reduce the GraphPi execution time by the highest speedup reported in FlexMiner (*i.e.,* 40×). We compare this FlexMiner performance to Mint by conservatively ignoring the execution time of phase 2, which provides a performance upper bound for this baseline.

VIII. Results

A. Performance Analysis

Benefit of search index memoization. To find the performance benefit of search index memoization, Fig. 10 compares the performance of Mint with and without applying this optimization with Mackey *et al.* [38]. This figure shows that, on average, the proposed optimization improves the performance of Mint from 91.6× to 363.1×. On average, the proposed optimization improves the performance of Mint by 4×. The reason behind this performance improvement is the reduction in memory traffic. Our evaluation shows that this optimization reduces the memory traffic by 2.8×, on average (up to 30.6× for mining M2 on stackoverflow).

This effect is more prominent for large datasets (wiki-talk and stackoverflow) because they access large neighborhood sets, and filter operations lead to severe under-utilization of memory resources (Fig. 7). Our further investigation shows that the sizes of the largest 10% of vertex neighborhoods, which benefit the most from search index memoization, in wiki-talk dataset are 14.9×–38.6× larger than the four smaller datasets, on average. Similarly, the largest 10% of vertex neighborhoods in stackoverflow are 2.6×–6.7× larger than the four smaller datasets, on average. Therefore, search index memoization is the most effective in large datasets that significantly reduces futile neighborhood fetches, improving overall performance. Large vertex neighborhoods in wiki-talk and stackoverflow datasets further underscore the value of this optimization.

Comparison with state-of-the-art CPU baselines. Fig. 11 compares the performance of Mint with four state-of-the-art software frameworks running on CPU. Mint outperforms Mackey *et al.* [38] by 363.1×, on average. Note that both baseline and Mint use a task–centric programming model. The high performance improvement of Mint over Mackey *et al.* is attributed to (a) converting task context updates to on-chip accesses, (b) domain-specific architecture design that efficiently executes the algorithm, and (d) search index memoization that significantly reduces memory traffic. The second bar (light blue color) shows the performance improvement of Mint over a software baseline that implements the search index memoization optimization. While search index memoization reduces the amount of work in the search phase, it comes at the cost of performing and additional search in software. As shown in Fig. 11, most of the performance benefit of proposed optimization in software is offset by the overhead of an additional search. The figure shows that Mint outperforms a CPU baseline that implements search index memoization by 305.9×, on average.

Mint also outperforms Paranjape *et al.* [53] by 2575.9×, on average. As shown in prior work [38], Paranjape *et al.* suffers redundant computation when the number of static subgraphs are higher than temporal motifs as it mines static subgraphs before resolving temporal constraints. Additionally, Mint benefits from an optimized programming model and domain-specific hardware design. The open-source implementation [52] of Paranjape *et al.* does not support M3 and M4; we limit our comparison to M1 and M2. PRESTO [61] is an approximate algorithm that samples temporal edges and runs exact mining algorithm as Mackey *et al.* on these edges. The goal of PRESTO is to achieve better scalability by mining motifs on a subset of edges. Fig. 11 shows that Mint, despite using an exact algorithm, outperforms PRESTO by 16.2×, on average. Because PRESTO is an approximate algorithm,

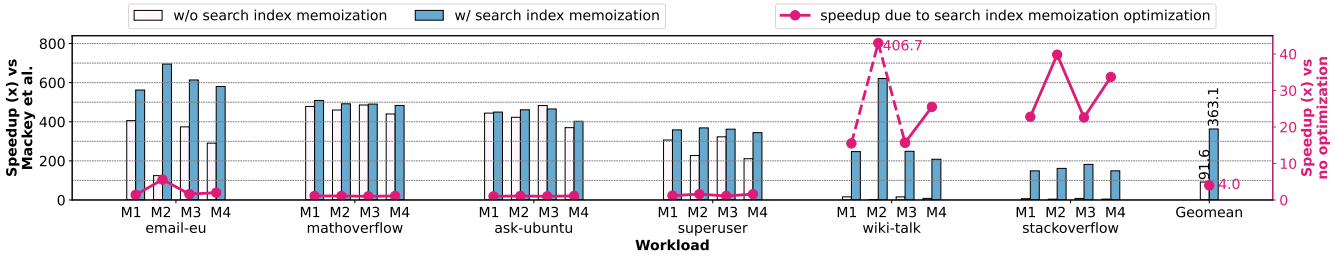


Figure 10. Performance improvement of Mint compared to Mackey *et al.* [38] and average memory bandwidth utilization, with and without the search index memoization optimization.

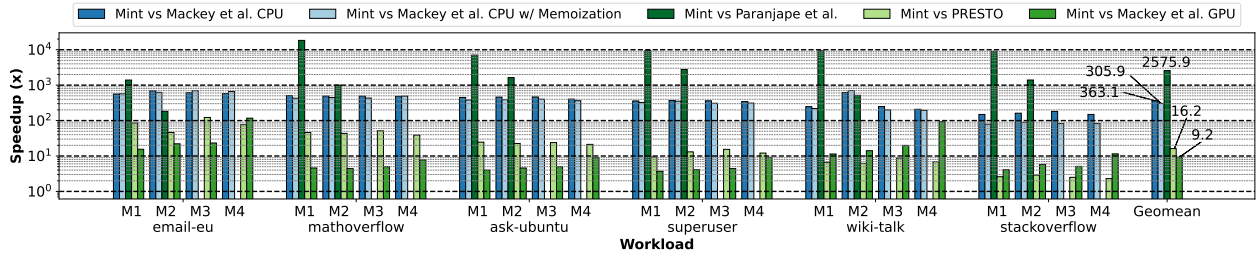


Figure 11. Performance improvement of Mint compared to Mackey *et al.* [38] CPU (without and with proposed optimization implemented in software), Paranjape *et al.* [53], PRESTO [61], and a GPU implementation of Mackey *et al.* The open-source codebase for Paranjape *et al.* only supports M1 and M2.

its resulting motif counts are mostly within 10% error of the actual value, whereas Mint mines all motifs. Because PRESTO also employs the same algorithm for mining motifs on a subset of edges, Mint can also accelerate PRESTO. *This results shows the value of hardware acceleration that can achieve both better quality results (by running an exact algorithm) and superior performance by designing its data path and memory subsystem to cater to an application's unique workload characteristics.*

Comparison with a GPU baseline. Fig. 11 shows that Mint significantly outperforms a GPU implementation of Mackey *et al.* [38] algorithm by 9.2×, on average. As discussed in §III-B, the temporal motif mining workload is bound by irregular memory accesses and control-flow instructions. While GPU improves the performance of this workload over a CPU baseline by offering massive parallelism and exploiting higher memory bandwidth, the GPU performance is limited due to the highly irregular nature of this workload leading to frequent thread divergence and non-coalesced memory accesses. Mint, on the other hand, further improves the performance of this workload by optimizing its data path to address unique workload characteristics of temporal motif mining. Furthermore, the peak memory bandwidth of a GPU is more than 3× the peak bandwidth of Mint. Due to high memory bandwidth utilization of this workload (§VIII-B), Mint can offer even higher speedup than reported compared to a GPU in an iso-bandwidth comparison. Moreover, Mint operates at 50× lower power (§VIII-C) than GPU that uses 250 W.

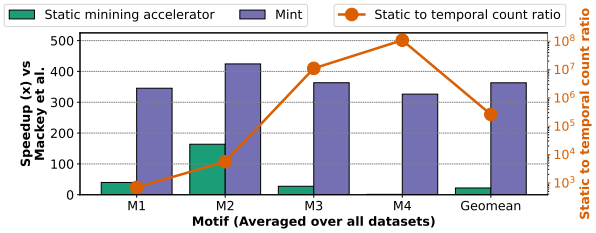


Figure 12. Performance of a static mining accelerator FlexMiner [13] and Mint compared to Mackey *et al.* The secondary y-axis shows the ratio between static to temporal motif counts. Results averaged over all datasets.

Comparison with a static graph mining accelerator. We further compare the performance of Mint with a static mining accelerator FlexMiner [13]. Fig. 12 shows that even by ignoring the temporal constraint resolution process, Mint achieves an order of magnitude better performance, on average, compared to FlexMiner. The figure further shows that the number of mined static graphs are significantly higher than the temporal motifs, which results in much more work for the static mining accelerator to perform. Temporal motif mining effectively prunes invalid subgraphs that do not meet temporal constraints, leading to significantly less work. *This result underscores the value of designing a temporal motif mining accelerator despite the availability of static mining accelerators [10], [12], [13], [15], [57], [71], [78].*

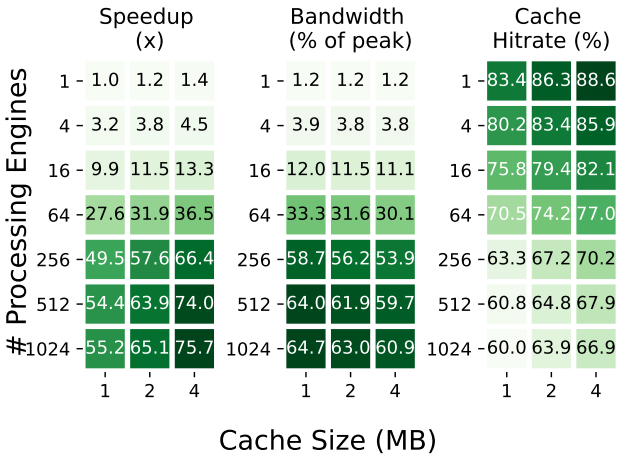


Figure 13. Sensitivity of performance (normalized to 1 processing engine 1 MB cache), bandwidth, and cache hit rate for mining M1 on a representative wiki-talk dataset.

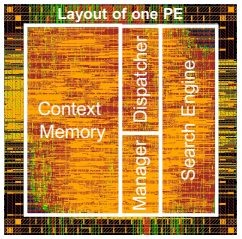
B. Sensitivity Analysis

To demonstrate the performance sensitivity of Mint, and the benefit of employing different hardware components, Fig. 13 shows how the performance, average memory bandwidth utilization, and cache hit rate changes for varying number of processing engines/PEs (a PE constitutes a context manager, a context memory instance, a dispatcher, and a search engine) and cache sizes for a representative workload of M1 mining on wiki-talk. The performance is normalized to a baseline configuration of 1 PE and 1 MB cache. The performance of Mint scales with the increase in compute resources and cache size. Specifically, by increasing the number of PEs by 1024× and cache size by 4×, the performance scales by 75.7×.

Adding compute resources enables exploiting more parallelism, and a larger cache size reduces the memory latency of the search phase. Scaling compute and memory resources also scales the memory bandwidth utilization. With fewer PEs, the workload is bound by the availability of compute resources that cannot saturate memory bandwidth. Adding compute resources shifts the workload from being compute bound to memory bound. Our evaluation shows that with 256 PEs, the workload slowly starts shifting from being compute to memory bound. With more PEs, Mint hardware expands multiple search trees in parallel, reducing the cache hit rate from 83.4% to 60%. However, increased memory and compute parallelism still improves overall performance of the workload. Additionally, with high concurrency, the workload starts experiencing cache port contention that constitutes 0.5% stall cycles for 1024 PEs, 4 MB cache.

C. Area and Power Analysis

Fig. 14 shows the layout of one PE, and the area and power consumption measurements of a full Mint design



Component	mm² Area	mW Power
Target Motif (1x)	< 0.001	6.8
Task Queue (1x)	< 0.01	< 0.1
Context Mem (512x)	4.98	265.0
64 KB cache (64x)	19.29	4698.2
Context Manager (512x)	0.36	18.9
Dispatcher (512x)	0.53	17.4
Search Engines (512x)	3.12	67.1
Crossbar (1x)	0.05	0.3
Total	**28.3 mm²**	**5.1 W**

*Power is measured at 1.6 GHz

Figure 14. Layout of one processing engine (PE) and area/power measurements of an entire Mint design.

based on post-synthesis results on a 28 nm technology node. The power results includes both leakage and dynamic power consumption. The dynamic power is averaged over all workloads. The table shows that Mint consumes an overall area of 28.3 mm^2 and 5.1 W power. A majority of area and power is consumed in on-chip SRAM caches that reduce the memory latency of search engines. As shown in Fig. 13, caches significantly improve the performance of Mint, therefore, their high share of area and power is justified. The cache consumes approximately equal amounts of power in dynamic and leakage energies. This is because of the multi-banked cache design, where all banks consume leakage power, whereas only one bank consumes dynamic power for each cache access. This multi-banked design, however, is desirable to reduce the performance hit due to cache contention.

IX. Related Work

Mint is the first work that designs a novel accelerator architecture for mining temporal motifs. Below, we compare Mint with the closest related works.

Software frameworks for static graph mining. Several software frameworks implement efficient graph mining algorithms on CPUs and GPUs. Early works [72] enumerate all possible subgraphs and then rule out invalid embeddings using isomorphism tests. Recent works [14], [25], [26], [39], [42], [65] avoid the expensive isomorphism tests and prune out redundant subgraphs. Other works reduce the memory consumption of intermediate subgraphs either by relying on SSD [75] or leveraging algorithmic techniques [16]. Approximate algorithms [24], [54], [58] aim to achieve better scalability on large graphs by mining subgraphs on a subset of edges. These frameworks, however, work for mining subgraphs in static graph inputs, and do not support temporal motif mining.

Software frameworks for temporal motif mining. As discussed in §II-C, several software frameworks have been designed to optimize temporal motif mining. Among these, a few works [32], [38], [53] propose exact algorithms, while others [37], [61], [74] achieve better scalability by sampling and mining only a subset of edges. Mint further optimizes performance of these software frameworks by proposing a

hardware accelerator. As shown in §VIII, Mint significantly outperforms state-of-the-art software baselines by 16–2576×.

Hardware acceleration for graph processing. Numerous acceleration techniques have been proposed to speed up graph processing on CPUs [4], [7], [46], [70], GPUs [62], and using dedicated accelerators [1], [2], [8], [9], [20], [45], [49], [55], [56], [66], [67], [77]. These works mostly focus on optimizing the irregular memory accesses of graph processing workloads. As discussed in §III-C, the memory access and computation patterns of temporal motif mining are unique (*e.g.,* search phase is not present in traditional graph workloads). Furthermore, a few prior accelerators [1], [55], [56] employ asynchronous execution models. While these design philosophies seem similar to Mint on the surface, the domain-specific nature of accelerators result in fundamental design differences. For example, Mint uniquely employs (a) no task insertion back into the task queue, (b) unique workload division between compute units, (c) domain-specialized context memory design, and (d) lack of prefetching and task coalescing (§VI-B) commonly employed in prior accelerators. Therefore, prior optimization techniques cannot be directly applied to accelerate temporal motif mining.

Hardware acceleration for static graph mining. Recent works propose hardware acceleration techniques for static graph mining, by either building domain-specific architectures [12], [13], [28], [57], [78] or by offloading the workload to near-data processing architectures [10], [15], [71]. While temporal motif mining is analogous to static subgraph mining, as shown in §III-C, their computation patterns are distinct. While acceleration techniques for static subgraph mining focus on optimizing set operations, temporal motif mining does not employ any set computation. Furthermore, Fig. 12 shows that using static mining accelerators to speed up temporal motif mining results in significantly more work and sub-optimal performance. Therefore, prior static graph mining accelerators cannot be directly used to support the intricate computation and memory access patterns of temporal motif mining.

Hardware acceleration for matrix operations. Matrix operations (both dense and sparse) have been heavily optimized using domain-specific accelerators [21], [50], [68], [80], GPUs [76], [79], FPGAs [36], and TPU [27]. While these techniques optimize matrix operations, as discussed in §II-D, temporal motif mining algorithms do not involve any matrix operations. In contrast, the studied workload employs unique operations on temporal graph data (*e.g.,* filtering temporal edge list, discovering new graph edges based on previously matched edges, and search backtracking) that cannot be expressed in terms of matrix operations efficiently. Therefore, prior matrix acceleration techniques cannot optimize the workload of temporal motif mining.

X. Conclusion

This paper presented Mint—a novel programming model and hardware accelerator for mining temporal motifs. The programming model divided the workload execution down in terms of three fundamental tasks and proposed a task–centric asynchronous execution model that unlocked massive opportunities for parallelism. We then proposed a domain-specific architecture that optimized its data path and memory subsystem design to best accelerate temporal motif mining using the proposed programming model. The hardware accelerator is motif and dataset-agnostic, and can be programmed to wwmine any arbitrary temporal motif. To further improve performance, we proposed search index memoization that significantly reduced memory traffic. Our evaluation demonstrated that Mint significantly outperformed state-of-the-art software frameworks by 16–2576× by using 28.3 mm^2 area.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This work is also supported by the United States - Israel BSF grant number 2020135.

References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1247–1262.

[2] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 134–145.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[4] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *ICS*, New York, NY, USA, 2016, pp. 39:1–39:11.

[5] U. Alon, "Network motifs: theory and experimental approaches," *Nature Reviews Genetics*, vol. 8, no. 6, pp. 450–461, 2007.

[6] A.-L. Barabási, "Network science," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1987, p. 20120375, 2013.

[7] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *HPCA*, Feb 2019, pp. 373–386.

[8] A. Basak, Z. Qu, J. Lin, A. R. Alameldeen, Z. Chishti, Y. Ding, and Y. Xie, "Improving streaming graph processing performance using input knowledge," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1036–1050. [Online]. Available: https://doi.org/10.1145/3466752.3480096

[9] N. Beckmann and D. Sánchez, "Cache-guided scheduling : Exploiting caches to maximize locality in graph processing," 2017.

[10] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, M. Copik, L. Kapp-Schwoerer, S. D. Girolamo, M. Konieczny, O. Mutlu, and T. Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[11] G. Bouritsas, F. Frasca, S. P. Zafeiriou, and M. Bronstein, "Improving graph neural network expressivity via subgraph isomorphism counting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[12] Q. Chen, B. Tian, and M. Gao, *FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators*. New York, NY, USA: Association for Computing Machinery, 2022, p. 43–55. [Online]. Available: https://doi.org/10.1145/3503222.3507730

[13] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2021, pp. 581–594.

[14] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proc. VLDB Endow.*, vol. 13, no. 8, p. 1190–1205, Apr. 2020. [Online]. Available: https://doi.org/10.14778/3389133.3389137

[15] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "Dimmining: pruning-efficient and parallel graph mining on near-memory-computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 130–145.

[16] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," p. 1357–1374, 2019. [Online]. Available: https://doi.org/10.1145/3299869.3319875

[17] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 175–184. [Online]. Available: https://doi.org/10.1145/1168857.1168880

[18] J. Glasser and B. Lindauer, "Bridging the gap: A pragmatic approach to generating insider threat data," in *2013 IEEE Security and Privacy Workshops*. IEEE, 2013, pp. 98–104.

[19] L. Hajdu and M. Krész, "Temporal network analytics for fraud detection in the banking sector," in *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*. Springer, 2020, pp. 145–157.

[20] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[21] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[22] P. Holme and J. Saramäki, "Temporal networks," *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[23] Y. Hulovatyy, H. Chen, and T. Milenković, "Exploring the structure and function of temporal networks with dynamic graphlets," *Bioinformatics*, vol. 31, no. 12, pp. i171–i180, 2015.

[24] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "{ASAP}: Fast, approximate graph pattern mining at scale," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 745–761.

[25] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: A pattern-aware graph mining system," 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387548

[26] K. Jamshidi and K. Vora, "A deeper dive into pattern-aware subgraph exploration with peregrine," *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, p. 1–10, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3469379.3469381

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang,

E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[28] O. Kalinsky, B. Kimelfeld, and Y. Etsion, "The triejax architecture: Accelerating graph operations through relational joins," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1217–1231. [Online]. Available: https://doi.org/10.1145/3373376.3378524

[29] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, p. 45–49, Jan. 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2414456

[30] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki, "Temporal motifs in time-dependent networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 11, p. P11005, 2011.

[31] L. Kovanen, K. Kaski, J. Kertész, and J. Saramäki, "Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences," *Proceedings of the National Academy of Sciences*, vol. 110, no. 45, pp. 18 070–18 075, 2013.

[32] R. Kumar and T. Calders, "2scent: An efficient algorithm to enumerate all simple temporal cycles," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1441–1453, 2018.

[33] M. Lahiri and T. Y. Berger-Wolf, "Structure prediction in temporal networks using frequent subgraphs," in *2007 IEEE Symposium on Computational Intelligence and Data Mining*, 2007, pp. 35–42.

[34] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[35] T. G. Lewis, *Network science: Theory and applications*. John Wiley & Sons, 2011.

[36] C. Y. Lin, N. Wong, and H. K.-H. So, "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[37] P. Liu, A. Benson, and M. Charikar, "A sampling framework for counting temporal motifs," *arXiv preprint arXiv:1810.00980*, 2018.

[38] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin, "A chronological edge-driven approach to temporal subgraph isomorphism," in *2018 IEEE international conference on big data (big data)*. IEEE, 2018, pp. 3972–3979.

[39] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, , and B. Wu, "Graphzero: Breaking symmetry for efficient graph mining," in *arXiv preprint arXiv:1911.12877*, 2019.

[40] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: Breaking symmetry for efficient graph mining," *arXiv preprint arXiv:1911.12877*, 2019.

[41] D. Mawhirter and B. Wu, "Automine: harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 509–523.

[42] ——, "Automine: Harmonizing high-level abstraction and high performance for graph mining," p. 509–523, 2019. [Online]. Available: https://doi.org/10.1145/3341301.3359633

[43] C. Meydan, H. H. Otu, and O. U. Sezerman, "Prediction of peptides binding to mhc class i and ii alleles by temporal motif mining," in *BMC bioinformatics*, vol. 14, no. 2. Springer, 2013, pp. 1–11.

[44] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[45] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 1–14.

[46] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates," in *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, October 2019.

[47] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to understand large caches," in *HP laboratories*, 2009.

[48] M. E. Newman, "The structure and function of complex networks," *SIAM review*, vol. 45, no. 2, pp. 167–256, 2003.

[49] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 166–177.

[50] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[51] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Physical Review E*, vol. 84, no. 1, p. 016105, 2011.

[52] A. Paranjape, A. R. Benson, and J. Leskovec, "Temporal motifs code in SNAP." [Online]. Available: https://snap.stanford.edu/temporal-motifs/code.html

[53] ——, "Motifs in temporal networks," in *Proceedings of the tenth ACM international conference on web search and data mining*, 2017, pp. 601–610.

[54] G. Preti, G. De Francisci Morales, and M. Riondato, "Maniacs: Approximate mining of frequent subgraph patterns through sampling," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 1348–1358.

[55] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 908–921.

[56] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "Jetstream: Graph analytics on streaming data with event-driven hardware accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1091–1105. [Online]. Available: https://doi.org/10.1145/3466752.3480126

[57] G. Rao, J. Chen, J. Yik, and X. Qian, "Sparsecore: Stream isa and processor specialization for sparse computation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 186–199. [Online]. Available: https://doi.org/10.1145/3503222.3507705

[58] T. Reza, M. Ripeanu, G. Sanders, and R. Pearce, "Approximate pattern matching in massive graphs with precision and recall guarantees," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1115–1131.

[59] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *arXiv preprint arXiv:2006.10637*, 2020.

[60] P. Rozenshtein and A. Gionis, "Mining temporal networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, p. 3225–3226.

[61] I. Sarpe and F. Vandin, "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, 2021, pp. 145–153.

[62] A. Segura, J.-M. Arnau, and A. González, "Scu: a gpu stream compaction unit for graph processing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 424–435.

[63] H. Shao, M. Marwah, and N. Ramakrishnan, "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, no. 1, 2013, pp. 1327–1333.

[64] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon, "Network motifs in the transcriptional regulation network of escherichia coli," *Nature genetics*, vol. 31, no. 1, pp. 64–68, 2002.

[65] T. Shi, M. Zhai, Y. Xu, and J. Zhai, *GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination*. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

[66] S. G. Singapura, A. Srivastava, R. Kannan, and V. K. Prasanna, "Oscar: Optimizing scratchpad reuse for graph processing," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[67] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543.

[68] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[69] N. Talati, D. Jin, H. Ye, A. Brahmakshatriya, G. Dasika, S. Amarasinghe, T. Mudge, D. Koutra, and R. Dreslinski, "A deep dive into understanding the random walk-based temporal graph learning," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 87–100.

[70] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," pp. 654–667, 2021.

[71] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. Blaauw, T. Mudge, and R. Dreslinski, "Ndminer: Accelerating graph pattern mining using near data processing," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2022.

[72] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," p. 425–440, 2015. [Online]. Available: https://doi.org/10.1145/2815400.2815410

[73] K. Tu, J. Li, D. Towsley, D. Braines, and L. D. Turner, "Network classification in temporal networks using motifs," *arXiv preprint arXiv:1807.03733*, 2018.

[74] J. Wang, Y. Wang, W. Jiang, Y. Li, and K.-L. Tan, "Efficient sampling algorithms for approximate temporal motif counting," in *Proceedings of the 29th ACM international conference on information & knowledge management*, 2020, pp. 1505–1514.

[75] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," p. 763–782, 2018.

[76] T. Yamaguchi and F. Busato, "Accelerating matrix multiplication with block sparse format and nvidia tensor cores," 2021.

[77] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 615–628. [Online]. Available: https://doi.org/10.1145/3352460.3358318

[78] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.

[79] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with gpu tensor cores," *Computers & Electrical Engineering*, vol. 88, p. 106848, 2020.

[80] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.