Journal of Parallel and Distributed Computing 191 (2024) 104903

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

DuMato: An efficient warp-centric subgraph enumeration system for GPU

Samuel Ferraz [a,b,*], Vinicius Dias [c], Carlos H.C. Teixeira [a], Srinivasan Parthasarathy [d], George Teodoro [a], Wagner Meira Jr. [a]

[a] Federal University of Minas Gerais (UFMG), Belo Horizonte, MG, Brazil
[b] Federal University of Mato Grosso do Sul (UFMS), Campo Grande, MS, Brazil
[c] Federal University of Lavras (UFLA), Lavras, MG, Brazil
[d] The Ohio State University (OSU), Columbus, OH, USA

ARTICLE INFO

Keywords:
Graph pattern mining
GPU
Irregular processing
Load-balancing
Enumeration paradigms

ABSTRACT

Subgraph enumeration is a heavy-computing procedure that lies at the core of Graph Pattern Mining (GPM) algorithms, whose goal is to extract subgraphs from larger graphs according to a given property. Scaling GPM algorithms for GPUs is challenging due to irregularity, high memory demand, and non-trivial choice of enumeration paradigms. In this work we propose a depth-first-search subgraph exploration strategy (DFS-wide) to improve the memory locality and access patterns across different enumeration paradigms. We design a warp-centric workflow to the problem that reduces divergences and ensures that accesses to graph data are coalesced. A weight-based dynamic workload redistribution is also proposed to mitigate load imbalance. We put together these strategies in a system called DuMato, allowing efficient implementations of several GPM algorithms via a common set of GPU primitives. Our experiments show that DuMato's optimizations are effective and that it enables exploring larger subgraphs when compared to state-of-the-art systems.

1. Introduction

The goal of Graph Pattern Mining (GPM) is to discover subgraphs that meet a set of criteria. For example, clique listing is a GPM task aiming to visit all *subgraphs* that represent a clique *pattern* with $k > 2$ vertices in a graph, and can be used to detect communities in graphs [36]. GPM algorithms are used in areas such as biology [43], social network analysis [15], among others. Given an input graph G, GPM algorithms enumerate subgraphs of G that match a given property. This property can be topological (e.g., clique, chordal, etc.) or statistical (e.g., pattern frequency [4]). GPM algorithms rely on a procedure called *subgraph enumeration*, which recursively combines subgraphs with their adjacency lists to produce larger subgraphs up to k vertices of an input graph.

Fig. 1 depicts one step of subgraph enumeration (extend) using a subgraph $s \leftarrow \{1, 2\}$ of an input graph. The adjacency lists of s are visited to generate a set of $extensions$, which correspond to vertices/edges that can be used to extend s and generate new subgraphs satisfying the given property. The same procedure is applied recursively for each extended subgraph. As such, subgraph enumeration deals with a combinatorial explosion in the number of subgraphs as the size of the visited subgraphs increases, which leads to long execution times and motivates the use of parallelism. For example, the small biological dataset *bio-diseasome* [41] containing 516 vertices and 1.2 K edges, around 4 TB of memory is required to store all induced subgraphs with ten vertices (assuming a 4-byte integer per vertex to store each subgraph).

Subgraph enumeration systems were proposed for CPU [13,27,45,37,47] and GPU [9,7,48,21], offering a good tradeoff between programmability and performance. These systems are supposed to provide an easy-to-use high-level framework that allows efficient implementations of different GPM algorithms within the same software environment. In order to use GPUs as a target platform and take full advantage of its massive parallelism, they try to mitigate the critical challenges in using GPU in the domain: *irregularity*, *combinatorial explosion* and *enumeration paradigm*.

Irregularity occurs due to the unpredictable cost for enumerating different subgraphs, which results from different sizes of adjacency lists of vertices in those subgraphs. The first consequence of irregularity is *thread divergence* when threads in a warp process different subgraphs. Second, there is also the lack of *memory coalescence* since parallel threads often need to access different adjacency lists. Third, is the inherent *load imbalance* in such computations. Since the processing costs of subgraphs (e.g. accessing adjacency lists) are not the

* Corresponding author.
E-mail addresses: samuel.ferraz@ufms.br, samuelbferraz@gmail.com (S. Ferraz).

https://doi.org/10.1016/j.jpdc.2024.104903
Received 30 June 2023; Received in revised form 13 December 2023; Accepted 11 April 2024
Available online 22 April 2024

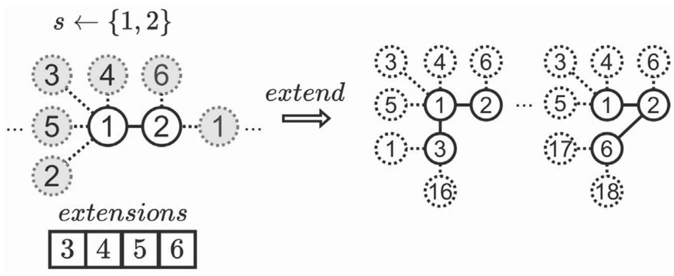


Fig. 1. One step of subgraph enumeration.

same, so are the subgraph enumeration loads assigned to particular threads.

Combinatorial Explosion happens due to the number of subgraphs that may be visited when the subgraph size k increases. The graph visitation strategy used directly impacts memory access patterns and parallelism. Breadth-First Search (BFS) and Depth-First Search (DFS) are the de-facto approaches to traverse graphs. BFS accesses the entire adjacency list of a subgraph, leading to coalesced memory access. However, it materializes all subgraphs throughout the enumeration, and consequently, the amount of memory required quickly grows with the subgraph size. This limits its use to visit small subgraph sizes [9,45]. The DFS reduces the memory demand and keeps a small portion of states during the enumeration. However, its irregular and strided memory requests may severely affect the GPU performance.

The *Enumeration Paradigm* (*pattern-oblivious* or *pattern-aware*) guides how subgraphs are visited during enumeration. The pattern-aware paradigm requires a target pattern as input to create a custom exploration plan [3,23] and visits only subgraphs matching that pattern, eliminating the need for graph isomorphism in the enumeration. Custom exploration plans reduce the number of intermediate states and the computational cost of enumeration but increase the complexity of using this paradigm for arbitrary patterns. The *pattern-oblivious* paradigm visits all subgraphs of a specific size regardless of pattern structure, thus eliminating the need of custom tuned exploration plans and recurrent visitations over the same input graph. As it visits all subgraphs, it has a higher computational cost than the pattern-aware paradigm for a single pattern. As the size of the enumerated subgraphs and the amount of patterns increase, the tradeoff between using these paradigms is non-trivial to quantify. Therefore, we argue that supporting both paradigms is an important feature.

This paper presents a GPM system called DuMato, which proposes execution strategies and optimizations to mitigate the challenges for efficient subgraph enumeration on GPUs. DuMato allows the efficient implementation of different GPM algorithms using a high-level API that supports both enumeration paradigms (pattern-aware and pattern-oblivious). DuMato addresses the irregularity with a subgraph exploration strategy that reduces memory requirements compared to BFS, enables better memory access coalescency and parallelism opportunities than DFS, and provides opportunities of data reuse throughout the enumeration. We also developed novel load-balancing strategies that improve the GPU utilization with low overhead and, consequently, the overall performance of the GPM algorithms developed in DuMato. This paper extends our previous work [18] with a detailed description of the solution and a more detailed related work; a redesign in the execution workflow with general-purpose core primitives to allow a new optimization that integrates the compaction phase of the DuMato with the filter phase; a novel load-balancing strategy; the inclusion of another GPM application (subgraph matching); the support for both enumeration paradigms (pattern-aware and pattern-oblivious). The experiments have also been carried out in more detail, including a new optimization formulation and an ablation analysis that studies the effect of parameters on performance. The main contributions of the work are summarized as follows:

- We propose the *DFS-wide traversal strategy* targeting GPU systems to reduce the memory demand and improve regularity. DFS-wide alternates between BFS and DFS to provide opportunities for memory coalescence with reduced memory demand. The DFS-wide modeling has a predictable affordable memory consumption and allows the reuse of intermediate states throughout the enumeration. DuMato enumeration phases have been modeled as *warp-centric steps* to minimize divergence. All phases were proposed based on only three core primitives, allowing a simplification in the execution workflow that improves the overall performance and simplifies the API. Our novel warp-centric design, along with the DFS-wide traversal strategy, achieved an average speedup of 67× vs DFS.
- We developed a low-overhead *warp-level load-balancing* strategy that uses the CPU to monitor and migrate load among GPU thread warps. We extended our previous load-balancing to propose a new strategy that uses the weight of each warp to redistribute workload better, and several jobs are allocated to warps to increase GPU's occupancy. We evaluated the load-balancing layer and the performance impacts when varying the parameters of the algorithm. This strategy leads to an average speedup of 19× vs the warp-centric design without load balancing and improves our previous load-balancing algorithm [18] by an average speedup of 1.40×.
- Underpinning the above efforts, we present a novel GPM runtime system called DuMato, targeting GPUs. DuMato proposes a general-purpose execution modeling and workflow for GPM on GPUs, and is the first subgraph enumeration system to support both enumeration paradigms (pattern-oblivious and pattern-aware). DuMato is typically an order of magnitude faster and has been able to mine larger subgraphs than other systems.

We have made our system publicly available through the following link: https://github.com/samuelbferraz/DuMato.

2. Background

This section introduces definitions used to represent GPM algorithms. Let $V(G)$ and $E(G)$ be, respectively, vertices and edges of a graph G. Without loss of generality, assume undirected graphs without labels. A *traversal* (Definition 2) represents an order that the vertices of a subgraph are visited. The subgraph enumeration is a combinatorial procedure that generates new subgraphs by visiting *neighborhoods* (Definition 1).

Definition 1. Given a subgraph S of a graph G, the **neighborhood** of S is the set of vertices $N(S)$ such that, given any vertex v_i in $V(G)$, v_i belongs to $N(S)$ iff v_i is not in $V(S)$ and is adjacent to at least one vertex in $V(S)$.

Definition 2. A **traversal** over a graph G is a list of its vertices, denoted as tr, and for any two values $tr[i]$ and $tr[j]$ where $i < j$, $tr[i]$ was visited before $tr[j]$ in the traversal.

Subgraph enumeration may visit distinct subgraphs with the same properties, and we say there is an *isomorphism* (Definition 3) between them. Two different traversals may visit the same subgraph S in different orders, and we say there is an *automorphism* (Definition 3) between them.

Definition 3. An **isomorphism** is a function that maps vertices of two graphs, G and H, in a one-to-one correspondence. It is called an isomorphism if for every edge between vertices in G, there is a corresponding edge between vertices in H. If the isomorphism between two graphs G and H is such that the vertices of G and H are the same, then it is called an **automorphism**.

Given a graph G and a subgraph S of G, one may reach S through different traversals. The only traversal allowed to visit a subgraph S in a graph G is known as the *canonical candidate*. Algorithms that filter canonical candidates are essential to avoid redundant computation throughout enumeration. To categorize subgraphs, GPM algorithms usually convert canonical candidates to a unique representation in a procedure known as *canonical relabeling*.

The subgraph enumeration procedure receives a graph G, a starting traversal tr, the target size k of the enumerated subgraphs, a property function P used during enumeration to keep only the traversals matching the desired property, and an output function A that produces results of the algorithm (e.g., a counter). One should provide specific P and A functions to implement a GPM algorithm.

Algorithm 1: Pattern-oblivious and pattern-aware paradigms.

```
1  void $PO(G, tr, k, P, A)$:
2    if($|tr| == k$):
3      $A(tr)$;
4      return;
5    $N' \leftarrow N(tr)$;
6    for each($candidate \in N'$):
7      $tr' \leftarrow tr.append(candidate)$;
8      $canonical \leftarrow isCanonical(tr')$;
9      $match \leftarrow P(tr')$;
10     if($canonical$ and $match$):
11       $PO(G, tr', k, P)$;
12
13 void $enumerate\_PO(G, k, P, A)$:
14   for each($v \in V(G)$):
15     $PO(G, \{v\}, k, P\ A)$;
16 void $PA(G, tr, k, EP)$:
17   if($|tr| == k$):
18     $A(tr)$;
19     return;
20   $N' \leftarrow \emptyset$;
21   for each($i \in EP[|tr|].adjacencies$):
22     $N' \leftarrow intersect(N', tr[i].adj)$;
23   for each($candidate \in N'$):
24     $sym \leftarrow breakSym(tr, candidate, EP)$;
25     $tr' \leftarrow tr.append(n)$;
26     if($sym$) : $PA(G, tr', k, P, EP)$;
27
28 void $enumerate\_PA(G, k, patterns, A)$:
29   for each($pattern \in patterns$):
30     for each($v \in V(G)$):
31       $PA(G, \{v\}, k, getEP(pattern), A)$;
```

Algorithm 1 depicts the implementation of subgraph enumeration for both the enumeration paradigms: *pattern-oblivious* (*enumerate_PO*) and *pattern-aware* (*enumerate_PA*). In both algorithms, we use the term *candidate* to represent vertices belonging to the neighborhood of the current traversal. In the pattern-oblivious approach (*PO* function), all vertices in the neighborhood of the traversal are used to create candidates regardless of the desired pattern. To eliminate automorphisms, custom canonicality-checking algorithms are applied in each candidate (line 8). Custom pattern functions are required to keep only traversals matching the pattern (line 9), and these functions usually rely on subgraph isomorphism tests. Subgraph enumeration continues for canonical candidates matching the pattern (line 10) as long as a traversal does not reach the limit size (line 2). In this paradigm, we need to enumerate all subgraphs starting from each vertex only once (lines 14-15), as all subgraphs targeting the desired property are visited.

The pattern-aware approach (*PA* function) receives an extra parameter: a pre-processed exploration plan (EP). An EP is an array with k vertices such that $EP[i]$ indicates which vertices of a traversal with i vertices must have their adjacency lists visited to generate candidates matching the desired pattern. In this paradigm, the candidates are generated using set-intersection operations [37] without the need of the pattern function P. Lines 21-22 generate the initial set of candidates using a subset of the neighborhood indicated by the exploration plan, and for each candidate (line 23), canonical candidates are kept applying custom straightforward comparisons (called *symmetry breaking rules*) of each candidate with specific vertices in the traversal (line 24). As the pattern-oblivious paradigm, subgraph enumeration continues for canonical candidates matching the pattern (line 26) as long as a traversal does not reach the limit size (line 17). The pattern-aware paradigm starts an enumeration from each vertex of the graph for each pattern modeling the desired property (lines 29-31), thus incurring in more iterations over the graph than the pattern-oblivious paradigm.

The computational cost of the pattern-oblivious paradigm does not depend on customized exploration rules. Besides, in order to visit all subgraphs of size k, it requires one call to the PO function starting from each vertex/edge of the input graph regardless of the amount of patterns searched. This approach has three major drawbacks: it visits adjacency lists that do not have chances of matching the pattern; the linear-time complexity of the general-purpose canonicality checking algorithms [45] applied over a high volume of candidates throughout enumeration; the need for constant isomorphism checking in each canonical candidate (the state-of-the-art isomorphism tools are designed for CPU [39,29]).

The pattern-aware approach visits only the adjacency lists that can generate candidates matching the pattern. Thus, it does not visit unnecessary adjacency lists. After the intersection between the restricted set of adjacency lists, the candidates generated are inherently matched to the pattern, thus not requiring subgraph isomorphism algorithms. Besides, the symmetry-breaking rules are constant-time comparisons between the candidates and the vertices. This approach has two significant drawbacks: its efficiency varies depending on the exploration plan, whose quality is hard to measure before the execution as the characteristics of the dataset influence it; given a set of patterns to be searched in an input graph G, this paradigm requires one call to the PA function starting from each vertex/edge of the input graph for each pattern, and as not all the patterns are necessarily present in the graph, some of these calls represent a waste of computational time.

3. Related work

A significant amount of GPM research has been devoted to architecture-conscious implementations targeting various parallel processors (CPUs, GPUs and other accelerators) [11,1,32,40,17,30,34,46,26,2,5,51,6]. However, these solutions do not provide a general-purpose environment that allows the design of custom programs for different application scenarios and thus, this section primarily reviews the literature on GPM *systems*.

GPM Systems for CPU. *Arabesque* [45] is one of the first GPM systems targeting distributed memory machines. It is *pattern-oblivious* and proposes a data structure (ODAG) to compress subgraphs in-memory to mitigate the memory demands of the BFS while it also employs load balancing. BFS and the pattern-oblivious design are frequently adopted together by out-of-core GPM systems. G-miner [8] and its successor G-thinker [49] are distributed frameworks that use a task-based strategy for accelerating out-of-core graph mining computations, but they lack high-level abstractions for improved programming experience. *RStream* [47] is a relational GPM system that relies on expensive join operations to perform subgraph enumeration. *Kaleido* [52] is an out-of-core system. It proposes a novel compact data structure to store intermediate enumeration states (an improvement over Arabesques's ODAG) with an I/O layer. The key contribution of *Kaleido* w.r.t. load imbalance is a strategy to predict the size of the subgraph candidates level by level and use this information to create subgraph partitions between threads. Although RStream and Kaleido are out-of-core, their design based on BFS limits the length of subgraphs feasible to be enumerated due to the inherent combinatorial explosion of GPM problems.

Fractal [13] is a distributed memory system to use a DFS strategy and to focus on programming productivity. DFS reduces the materialization of the intermediate states of the enumeration and, consequently, the memory requirements. In order to mitigate load imbalance, an intra- and inter-machine work-stealing mechanism is proposed. Whenever the patterns of subgraphs are known apriori, the search can be optimized via specific (pattern-oriented) execution plans.

LIGHT [44] adopts the pattern-aware approach and accelerates the computation with vectorized set intersection operations and by avoiding redundant computations. Besides the fact that LIGHT is designed specifically for subgraph matching task, it remains unclear how to handle and to compose more complex applications relying on customized filtering conditions and/or multi-pattern subgraph exploration. *AutoMine* [37] fills this gap by proposing an automated code generation for custom patterns that explicitly leverages loop-invariant properties of nested loops that arise from pre-determined exploration plans. *Peregrine* [27] proposes an interface that allows high-level programming of GPM algorithms. It incorporates several optimizations such as avoiding redundant set operations [44] and minimum vertex cover matching [31]. Although Peregrine proposes a minimalist static load balancing scheme for shared-memory machines, it is unclear how to scale the strategy to GPUs.

GraphPi [42] and *GraphZero* [38] are pattern-aware systems that propose improvements for nested-loop-based (e.g., *AutoMine* [37]) implementations of GPM algorithms. *GraphPi* handles a different challenge which is the cost of symmetry-breaking conditions used by exploration plans in pattern-aware systems. *GraphZero*, on the other hand, proposes improvements to the code generation model for GPM programs and thus. Both systems are *DFS* and exhibit a low-memory footprint, but neither optimize memory access or propose dynamic load balancing schemes appropriate for massive parallel systems on GPU for skewed workloads such as GPM problems. Pattern-aware GPM systems must enumerate querying patterns apriori and generate explóration plans individually for each pattern (e.g., in motif counting of subgraphs with 8 vertices there are $11,117$ different patterns), which usually represents non-negligible overhead. To mitigate this issue, *SumPA* [19] proposes to merge generated patterns (and exploration plans) according to their similarities to reduce redundant computation. This solution, however, has limited impact.

GPM Systems for GPU. Pangolin [9] is the first GPM system to leverage GPUs. It uses a pattern-oblivious enumeration with BFS, which enables a Bulk Synchronous Parallel (BSP) model where subgraphs are materialized on GPU's memory and redistributed among processing units at each step. Because of the BFS high memory demands, Pangolin can only search for small subgraphs.

Most pattern-aware GPUs systems handle a more simple and fundamental task in which the goal is to match a query graph (pattern) against the data graph. PBE [20] proposes an exploration for the scenarios where the data graph does not fit into GPU's memory. PBE accomplishes this by using a customized graph partitioning scheme. VSGM [28] is build upon PBE's ideas and improves it by optimizing partitions (bins) via fast heuristics that allow overlapping processing and partition generation. PBE and VSGM do not directly handle the optimization of memory access on GPU neither the imbalance challenge inherent of subgraph enumeration. Thus, they can not enumerate *larger* subgraphs. However, our efficient GPU enumeration design could be enhanced by incorporating these partitioning schemes to handle larger graphs.

RPS [21] is a pattern-aware BFS system that leverages reuse of set intersections. This work is also complementary and orthogonal to ours – in fact, the routines proposed by RPS to optimize the search space exploration of a specific pattern can be implemented through our optimized warp-centric primitives, ensuring coalesced memory accesses and balanced executions.

PARSEC [14] implements subgraph enumeration via pattern-aware and a hybrid BFS/DFS exploration. It matches the first two vertices of a pattern using BFS to materialize a set of traversals and generate parallel GPU tasks. Each parallel task is responsible for independently enumerating a subset of subgraphs via a DFS exploration. Although this static workload generation and distribution may be effective for small subgraphs, imbalance becomes critical as larger subgraphs are processed. Our load balancing strategy is dynamic and do not suffer from this problem, as it reacts to imbalance at run-time.

STMatch [48] considers a slightly different subgraph enumeration problem that does not break symmetries. Rather than eliminating automorphisms, STMatch visits the same subgraph several times, and unnecessary results are removed using the theoretical multiplicity of the pattern. It proposes a hierarchical work-stealing mechanism to balance the load on one or multiple GPUs.

G2Miner [7] is a recent pattern-aware GPU system, a successor (and improved version) of Pangolin [9]. G2Miner provides a warp-centric scheme to perform *set-intersection* operations and improve divergences and memory coalescence, but only this operation is addressed and the rest of the enumeration pipeline does not benefit from it. Thus, gains obtained with this strategy are small compared to their baseline implementation (2×). For example, our warp-centric enumeration scheme achieved an average speedup of 67× due to increased memory coalescence and lockstep execution. Besides, the system inherits all limitations of pure pattern-aware systems.

In general, while GPU GPM systems attained notable performance gains, these works have not fully addressed all critical challenges of subgraph enumeration targeting this architecture (combinatorial explosion, memory uncoalescence, divergences, load imbalance and flexible enumeration paradigm). The DuMato system proposed here deals with all these challenges and can efficiently mine large subgraphs. Moreover, DuMato is extensible in the sense that it is build over a set of representative GPU primitives, reducing the effort to incorporate multiple optimizations in the same system (even from different paradigms) and making the modeling of algorithms more productive.

4. Strategies for efficient high-level subgraph enumeration on GPUs

This section presents our optimization strategies to mitigate the main challenges for subgraph enumeration on GPUs: the high memory demand generated by *combinatorial explosion*; the memory uncoalescence, divergences and load imbalance generated by *irregularity*; the flexibility in the choice of the *enumeration paradigm*. The *DFS-wide traversal strategy* ensures that memory demand is bounded and adjusted based on the choice of the enumeration paradigm. *DuMato*, our high-level *warp-centric* subgraph enumeration system, also mitigates *memory uncoalescence* and *divergences* by providing regular execution and memory access patterns, and provides a flexible API to allow the implementation of GPM algorithms using both enumeration paradigms (*pattern-oblivious* and *pattern-aware*). Our *warp-level load balancing* mitigates the load imbalance.

4.1. DFS-wide enumeration strategy

DFS-wide is our novel strategy to traverse a graph on the GPU and keep the intermediate states needed for both enumeration paradigms. DFS-wide alternates between BFS and DFS phases to provide regular execution with affordable memory use. Fig. 2 depicts the enumeration lattice generated to visit the subgraph $\{2,3,4,6\}$ using BFS, DFS, and DFS-wide. In BFS, all intermediate traversals throughout enumeration are materialized. This allows regular memory accesses, but the combinatorial explosion of stages makes BFS memory demand too high. DFS generates the minimum amount of intermediate states. Despite its low memory consumption, the memory access pattern of DFS is more sparse and deteriorates locality.

Different from other systems that use a DFS-like traversal strategy [14,48], the memory consumption of our DFS-wide strategy is ad-

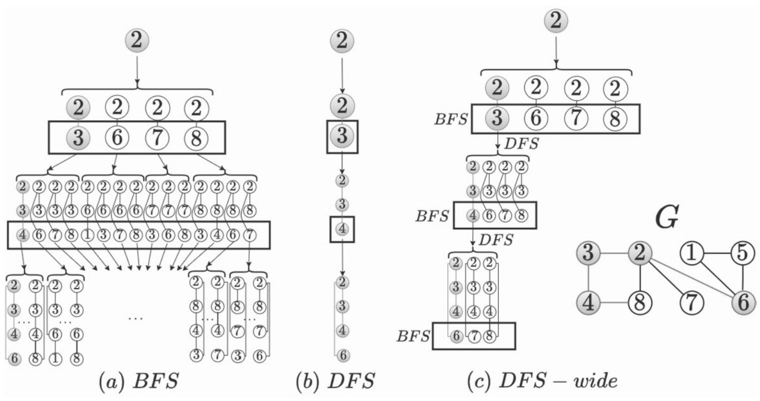


Fig. 2. BFS, DFS, and DFS-wide strategies.

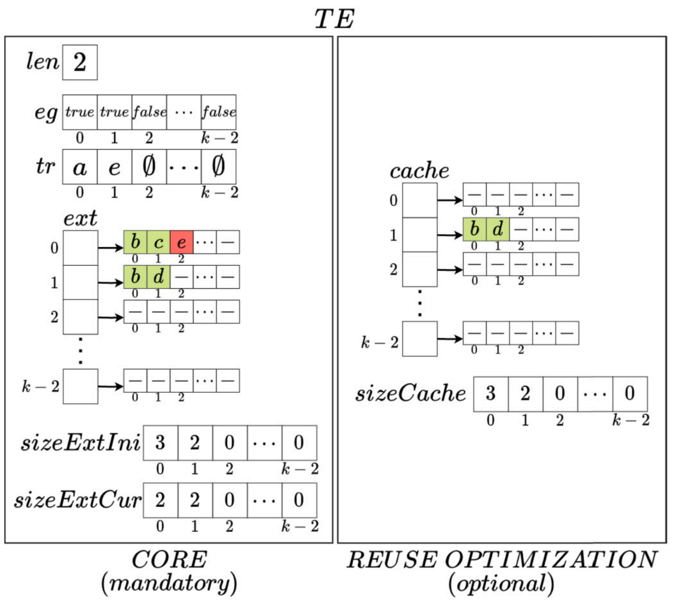


Fig. 3. Traversal Enumeration data structure used for DFS-wide exploration.

justed according to the enumeration paradigm chosen for enumeration. Besides, it provides a cache that enables the reuse of intermediate states, which constitutes the basis for many optimizations proposed by state-of-the-art pattern-aware systems. To support such flexibility of paradigms while still ensuring an efficient use of the GPU, we propose an efficient data structure for maintaining the state of a DFS exploration of traversals. Fig. 3 depicts this data structure considering a snapshot using the traversal $tr = \{a, e\}$.

The traversal enumeration data structure (TE) Both paradigms use a structure called TE (*Traversal Enumeration*) to keep track of all enumeration state required by the DFS-based subgraph exploration, i.e., the set of vertex/edge *extensions* used to produce larger traversals (subgraphs). The attribute len stores the current size of the traversal. The attribute tr is an array with $k-1$ integers to store the ids of the vertices in the current traversal (the last id of the traversal is not materialized to save memory, but it is combined with the remaining of the traversal for aggregation as usual). The attribute eg is an array with $k-1$ booleans such that $TE.eg[i]$ indicates whether the extensions of the traversal $TE.tr[0\cdots i]$ have been generated.

The extensions are modeled by a set of $k-1$ arrays such that $TE.ext[i]$ ($i < k-2$) stores the set of extensions generated by the traversal $TE.tr[0\cdots i]$. For example, $TE.ext[1]$ stores the extensions generated by $TE.tr[0,1]$ ($\{a,e\}$). The $sizeExtIni$ array contains $k-1$ positions such that any $TE.sizeExtIni[i]$ stores the initial number set of extensions in $TE.ext[i]$, and $TE.sizeExtCur[i]$ stores the actual amount of extensions in $TE.ext[i]$. We use the last extension of $TE.ext[i]$ to move forward in the enumeration, and the removal of this extension is implemented as a decrement in the $TE.sizeExtCur[i]$. The information on $TE.sizeExtInit[0\cdots i]$ ensures that original extensions can be reused and retrieved even after removals indicated by decrements on $TE.sizeExtIni[0\cdots i]$. Indeed, for many patterns, it is beneficial to cache intermediate extensions to avoid repeated and redundant accesses to the adjacency lists of the traversal vertices $TE.tr[0\cdots k-2]$. To enable this optimization, we keep a set of $k-1$ arrays ($cache$) to store this information whenever the exploration plan of a pattern indicates the possibility of reuse (an example is presented in Algorithm 3).

Table 1
Worst-case memory consumption of enumeration paradigms.

Data structure	Paradigm	
	Pattern-oblivious	Pattern-aware
TE.len	1	1
TE.tr	$k-1$	$k-1$
TE.eg	$k-1$	$k-1$
TE.ext	$\sum_{i=1}^{k-1} i \times maxd(G)$	$(k-1) \times maxd(G)$
TE.cache	–	$(k-1) \times maxd(G)$
TE.sizeExtIni	$k-1$	$k-1$
TE.sizeExtCur	$k-1$	$k-1$
total	$O(k^2 \times maxd(G))$	$O(k \times maxd(G))$

The storage of the intermediate extensions (ext) and its caching ($cache$) dominate the memory cost of the TE data structure. The pattern-oblivious paradigm does not use caching and the extensions are generated regardless of a pattern. For example, $TE.ext[1]$ is allocated to have enough space to store the worst-case scenario where the two vertices in the traversal have the maximum degree of the graph ($maxd(G)$). Thus, $TE.ext[1]$ has $2 \times maxd(G)$ positions. This allocation requirement is repeated until $TE.ext[k-2]$, which leads to a total amount of $\sum_{i=0}^{k-2} i \times maxd(G)$ integer positions to store the extensions in this paradigm. In the pattern-aware paradigm, the extensions are extracted from the intersection of adjacency lists. Thus it requires at most $maxd(G)$ positions for any $TE.ext[i]$, and the same amount is required for $TE.cache[i]$. Thus, the total amount of integer positions to store the extensions in this paradigm is $\sum_{i=0}^{k-2} 2 \times maxd(G)$. The $sizeExtInit$, $sizeExtCur$ and $sizeCache$ arrays contain $k-1$ positions.

Table 1 depicts the worst-case memory consumption of one enumeration task using our DFS-wide strategy for each enumeration paradigm. DFS-wide fulfills the memory requirements to execute subgraph enumeration on a GPU with 12 GB of memory to visit subgraphs up to 31 and 11 using the pattern-aware and pattern-oblivious paradigms, respectively, for any dataset with maximum degree up to 16 K (e.g., LiveJournal, Pokec) and using the recommended number of parallel enumeration tasks presented in Section 5 (102400 threads).

Table 2
DuMato API.

Functions	Phase	Scope
`[CT]` *control(TE)*	Control	Algorithm-independent
`[MV]` *move(TE)*	Move	
`[EX]` *extend(TE, m_ext, m_cache, m_adj, op)*	Extend	Algorithm-specific
`[FL]` *filter(TE, P, args)*	Filter	
`[A1]` *aggregate_counter(TE)*	Aggregate	
`[A2]` *aggregate_pattern(TE)*		
`[A3]` *aggregate_store(TE)*		
`[CC]` *cache(TE)*	Extend	Optimization

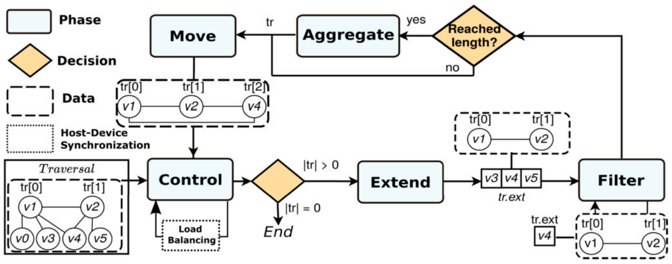


Fig. 4. DuMato execution workflow.

4.2. DuMato: an efficient high-level subgraph enumeration system for GPUs

We describe in this section our subgraph enumeration system called DuMato, that supports high-level customizable implementations of GPM algorithms on GPU. DuMato wraps around the DFS-wide strategy and a set of novel strategies and optimizations designed for efficient GPM processing on GPU.

4.2.1. Execution workflow and programming API

DuMato adopts the *filter-process* model [45] depicted on Fig. 4. The traversal tr is the input for the *Control* phase, which decides whether the enumeration should proceed or stop ($|tr| = 0$). In case it proceeds, the *Extend* phase assigns a subset of $N(tr)$ as its initial set of valid extensions. The *Filter* phase is an optional phase that keeps only the extensions that fulfill the desired property P. We optimized our previous execution workflow [18] such that invalid extensions are removed from the extensions array during the *Filter* phase, thus eliminating the *Compact* phase from the workflow. This optimization reduces the number of iterations through the extensions array and improves the performance. If the traversal tr reaches size k, the *Aggregate* phase produces the output of the algorithm ($|tr| = k$). The *Move* phase receives tr and decides whether to move forward (there are unprocessed extensions in tr) or backward (all extensions of tr were processed). The output of this phase is a modified version of tr that serves as input to the *Control* phase. The enumeration continues until all traversals that can be generated from the input traversal tr are visited. The primitives of *extend*, *filter*, *aggregate*, and *move* phases are implemented within the same kernel call. Device-wide synchronizations are required only by the *control* phase whenever it detects the need for a load-balancing step. In that case, the CPU copies the enumeration data structures and reorganizes them by applying our load-balancing algorithm, copying the data back to GPU, and rescheduling the enumeration kernel back to GPU.

Table 2 presents DuMato's API with functions categorized per workflow phase. Any GPM algorithm uses *Control* and *Move* to manage the workflow cycle. Thus, they are *algorithm-independent*. The other phases are *algorithm-specific* as they are optional or need parameters depending on the algorithm. Each function receives the TE argument (with runtime information about active traversals) and additional parameters. The *Control* phase (`[CT]`) allows runtime checking to determine whether the enumeration of a traversal should continue. The *Move* phase (`[MV]`) uses the size of current traversal and its extensions to decide moving forward or backward.

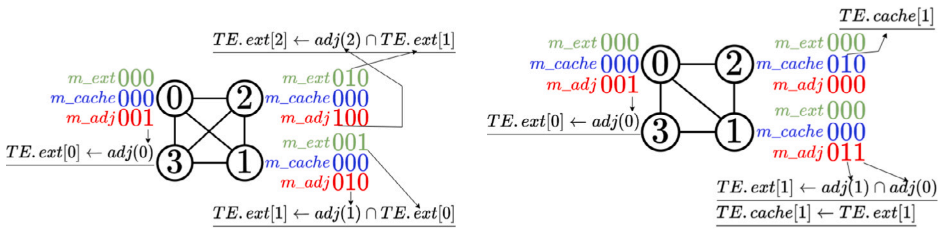
The *Extend* phase (`[EX]`) receives three masks: m_ext, m_cache and m_adj. Given any $i < k - 1$, the *i-th* lowest bit of m_adj, m_ext and m_cache indicates whether $TE.ext[i]$, $TE.cache[i]$ and the adjacency of $TE.tr[i]$ will be used to generate the current extensions, respectively. The last argument (op) is a function pointer that receives TE, m_ext, m_cache and m_adj, and performs the needed operations to generate the current extensions using the required arrays. We provide two standard op functions: $union$ (for pattern-oblivious algorithms) and $intersect$ (for pattern-aware paradigms).

The *Filter* phase (`[FL]`) receives a user-defined function pointer P, which models the property P defined in terms of subgraphs, along with its arguments ($args$). Filter calls the P function for each extension of the current traversal and removes those that do not satisfy P. We may use P to design custom subgraph filters for canonical candidate generation [45], density [35], subgraph matching [22], among others. The remaining functions (`[A1]`, `[A2]`, and `[A3]`) produce the outputs of the algorithms. These functions count the valid traversals visited throughout enumeration (`[A1]`), the number of valid traversals visited for each pattern (`[A2]`), or store the visited traversals for further downstream processing [12,25] (`[A3]`). We also provide an utility function $cache(TE)$ (`[CC]`), which may be called at any point of the execution workflow to copy the current set of extensions ($TE.ext[TE.len - 1]$) into the cache ($TE.cache[TE.len - 1]$) for further reuse.

4.2.2. Use case algorithms developed in DuMato

This section presents three representative GPM algorithms in DuMato: *clique counting*, *subgraph matching* and *motif counting*. *Clique counting* counts the number of cliques with k vertices in G. It represents algorithms that search for custom patterns with direct reuse of the extensions. Fig. 5a depicts the exploration plan used to visit cliques with 4 vertices. The vertex numbers represent the order in which the pattern vertices will be matched in the input graph. Since all vertices in a clique are connected, the order we visit them does not change the efficiency of enumeration. Each vertex has three masks associated with the use of extensions, cache and adjacency lists: m_ext, m_cache and m_adj. Given any $i < 3$, the enumeration aims to generate $TE.ext[i]$ (underlined in Fig. 5a) using the extensions, cache and adjacency lists indicated by the associated masks. For the clique pattern, in order to generate any $TE.ext[i]$, we need the computed extensions from the previous traversal $TE.ext[i-1]$ and the adjacency of the new vertex $TE.tr[i]$. As these sets are the only needed to generate the current extensions in clique, there is no need for copying $TE.ext[i - 1]$ to cache and the m_cache masks are set to zero. For example, to generate $TE.ext[1]$, the bit $m\_ext[0]$ is set to indicate the use of $TE.ext[0]$ and the bit $m\_adj[1]$ is set to indicate the use of the adjacency of $TE.tr[1]$. The set intersection operation is applied to generate any $TE.ext[i]$ up to the desired size.

Algorithm 2 presents the implementation of *clique counting*, and depicts the skeleton code used to implement GPM algorithms on DuMato. It is implemented in a loop-based so that the enumeration of traversals continues while the termination condition has not been reached (line 5). Extensions are generated whenever necessary (line 7), aggregation primitives are called when the size of visited subgraphs reaches the target size (line 11), and at the end of each loop iteration (line 13) DuMato moves to the next recursion step (forward/backward). These are common steps in GPM algorithms. The rest of the code is algorithm-specific


$TE.ext[2] \leftarrow adj(2) \cap TE.ext[1]$
m_ext 000
m_cache 000
m_adj 001
m_ext 010
m_cache 000
m_adj 100
m_ext 001
m_cache 000
m_adj 010
$TE.ext[0] \leftarrow adj(0)$
$TE.ext[1] \leftarrow adj(1) \cap TE.ext[0]$
0
1
2
3
$TE.cache[1]$
m_ext 000
m_cache 000
m_adj 001
m_ext 000
m_cache 010
m_adj 000
m_ext 000
m_cache 000
m_adj 011
$TE.ext[0] \leftarrow adj(0)$
$TE.ext[1] \leftarrow adj(1) \cap adj(0)$
$TE.cache[1] \leftarrow TE.ext[1]$

(a) 4-cliques (b) chordal square

Fig. 5. DuMato's representation of pattern-aware exploration plans.

(marked with icon 👤). Lines 2 – 4 set the masks of clique counting to generate any $TE.ext[i]$. The *Extend* phase (line 8) intersects the required sets to produce the current extensions, generating only those that are cliques. The filter implements the symmetry-breaking rule to eliminate automorphism (line 9). For cliques, the symmetry breaking rule ensures that new vertex added to the traversal must be greater than the last one. The Aggregate phase (line 11) increments the counting by accumulating the size of the extensions array.

Algorithm 2: Clique counting algorithm.

```
1     void clique_counting_4(TE)
2  👤   m_ext ← [000, 001, 010];
3  👤   m_cache ← [000, 000, 000];
4  👤   m_adj ← [001, 010, 100];
5      while(control(TE)):
6        last ← TE.len − 1;
7        if(!TE.eg[last]):
8  👤       extend(TE, m_ext[last], m_cache[last], m_adj[last], &inter);
9  👤       filter(TE, &lower_than, TE.tr[last]);
10       if(TE.len == k − 1):
11 👤       aggregate_counter(TE);
12       move(TE);
```

The *subgraph matching* represents GPM algorithms that search for custom patterns which allow different exploration plans that may take advantage of reuse through caching of snapshots of the extensions set. In order to depict its implementation on DuMato, we use the chordal pattern with 4 vertices, and Fig. 5b shows the exploration plan adopted. The generation of $TE.ext[0]$ requires only the adjacency list of $TE.tr[0]$, thus only $m\_adj[0]$ is set. The generation of $TE.ext[1]$ requires the intersection of the adjacency lists of $TE.tr[0]$ and $TE.tr[1]$, thus the bits $m\_adj[0]$ and $m\_adj[1]$ are set. Before applying symmetry-breaking to $TE.ext[1]$, we copy this set to $TE.cache[1]$, as it will be reused to generate $TE.ext[2]$. The generation of $TE.ext[2]$ requires again the intersection of the adjacency lists of $TE.tr[0]$ and $TE.tr[1]$, which was cached previously. Thus, we need only the set $TE.cache[1]$ to generate $TE.ext[2]$, indicated by the bit $m\_cache[1]$. The code is presented in Algorithm 2 for the chordal pattern with 4 vertices. The masks (lines 2-4) are set according to the exploration plan in Fig. 5b. The set intersection operation of specific sets guarantee only extensions matching the chordal pattern (line 8). We call the cache primitive to store the extensions of $TE.ext[1]$ for reuse in $TE.ext[2]$. The only filter needed is the one that implements the symmetry-breaking rule to eliminate automorphisms for the chordal pattern (line 10).

The *motif counting* algorithm (Algorithm 4) counts the occurrence of each possible pattern with k vertices in an input graph G, thus representing GPM algorithms that search for multiple patterns. Note that, different from the pattern-aware subgraph matching, this implementation is pattern-oblivious and does not rely on custom GPU kernels for each pattern, thus the same kernel can be used for any k. As any pattern may be visited during enumeration, there is no reuse of intermediate extensions, and all masks in m_ext (line 2) and m_cache (line 3) are set to zero. Given any $i < k - 2$, the mask in $m\_adj[i]$ (line 4) indicates

Algorithm 3: Subgraph matching of the chordal square.

```
1     void matching_chordal_4(TE)
2  👤   m_ext ← [000, 000, 000];
3  👤   m_cache ← [000, 000, 010];
4  👤   m_adj ← [001, 011, 000];
5      while(control(TE)):
6        last ← TE.len − 1;
7        if(!TE.eg[last]):
8  👤       extend(TE, m_ext[last], m_cache[last], m_adj[last], &inter);
9  👤       if(TE.len == 1) : cache(TE);
10 👤       filter(TE, &symmetry_chordal_4, [ ]);
11       if(TE.len == k − 1):
12 👤       aggregate_pattern(TE);
13       move(TE);
```

that all the adjacency lists of vertices in $TE.tr[0 \cdots i]$ are used to generate $TE.ext[i]$. In the case of the current extensions have not been generated (line 7), the *Extend* phase of the algorithm (line 8) is pattern-oblivious and performs the union of the adjacency lists of all vertices in the traversal to produce the current set of extensions. The *Filter* phase is implemented in one step, and the function pointer *is_canonical* (called in line 9) keeps only the canonical candidates using a generic canonical filtering algorithm [45] to eliminate automorphisms. Finally, if the traversal reaches the size of $k - 1$ (line 10), the Aggregate phase (line 11) converts each extension to its pattern and accumulates the counting for each pattern found.

Algorithm 4: Motif counting algorithm.

```
1     void motif_counting(TE)
2  👤 m_ext ← [00...00, 00...00, ⋯ , 00...00];
3  👤 m_cache ← [00...00, 00...00, ⋯ , 00...00];
4  👤 m_adj ← [00...01, 00...11, 00...111 ⋯ , 11...11];
5      while(control(TE)):
6        last ← TE.len − 1;
7        if(!TE.eg[last]):
8  👤       extend(TE, m_ext[last], m_cache[last], m_adj[last], &union);
9  👤       filter(TE, &is_canonical, []);
10       if(TE.len == k − 1):
11 👤       aggregate_pattern(TE);
12       move(TE);
```

4.2.3. *Warp-centric design*

The warp-centric programming model [24] is used in irregular algorithms to improve their execution regularity. In our design, a warp receives a traversal for processing, and threads within a warp alternate between SIMD and SISD phases throughout the execution workflow. Our goal with this model is to minimize execution divergence and to exploit the opportunities of parallelism and regular memory access enabled by the DFS-wide. The phases of DuMato's enumeration workflow were designed based on three main warp-centric primitives: *find_one*, *find_many*, and *write*, discussed next.

Core primitives. In essence, the core primitives are used to search for values in the adjacency lists and extensions, and are used during the enumeration phases and to implement specific *op* functions (depicted in Table 2). The *find_one* is employed when threads within a warp need to find whether the same value x is in an array v (Algorithm 5). Our default implementation has a linear-time complexity, as it does not assume an ordering of the input array. However, we also provide a log-time implementation whenever the array is ordered, thus reducing the overall complexity. The most important use of *find_one* primitive is during the subgraph induction, when threads within a warp induce a traversal and need to check whether one extension is in each adjacency list of the vertices in the traversal. The variable *found_local* stores whether the x was found in v by the current thread, and variable *found_global* stores whether any thread within a warp found x in v. The main loop (line 3) iterates through v in parallel (32 is the warp size) and each thread within a warp receives a different value v to compare with x using coalesced memory requests to access v (line 4). *Any_sync* (line 5) is a warp exchange primitive used to exchange variable *found_local*. In case any *found_local* variable is not 0, *any_sync* returns 1 for all threads within a warp and sets 1 to *found_global* for all threads. Otherwise, *found_global* is set to 0 and the search continues.

Algorithm 5: Primitive *find_one*.

```
1 $int\ find\_one(x, v, start, end):$
2   $found\_local \leftarrow found\_global \leftarrow 0;$
3   $for\ (pos \leftarrow start + lane;\ pos < end\ \&\&\ !found\_global;\ pos\ += 32):$
4     $found\_local \leftarrow v[pos] \leftarrow x;$
5     $found\_global \leftarrow any\_sync(found\_local);$
6   return $found\_global;$
```

The *find_many* is used when threads within a warp need to find different values in an array v (Algorithm 6). This primitive is used by warps to find values in the extensions and in the traversal in parallel, mainly in the extend and filter phases. The algorithm is similar to *find_one*, but with two crucial differences: variable *found_global* stores a mask such that the i-th bit stores whether the i-th thread in the warp has already found its value in v, and is built using the *ballot_sync* warp exchange primitive; the main loop continues for all threads within a warp as long as there is at least one thread that still has not found its value in v. Although we could propose a log-time implementation for *find_many* in the cases where the input array is ordered, this would generate plenty of extra memory transactions, as each thread of the warp searches for a different value and would demand different regions of the input array at the execution. Thus, for this primitive, we provide only a linear-time implementation.

Algorithm 6: Primitive *find_many*.

```
1 $int\ find\_many(value, v, start, end):$
2   $found\_local \leftarrow found\_global \leftarrow 0;$
3   $pos \leftarrow start;$
4   $for\ (;\ pos < end\ \&\&\ found\_global\ != \text{0xffffffff};\ start{+}{+}):$
5     $found\_current \leftarrow v[pos] == value;$
6     $found\_local \leftarrow found\_local\ ||\ found\_current;$
7     $found\_global \leftarrow ballot\_sync(found\_local);$
8   return $found\_local;$
```

The *write* primitive (Algorithm 7) is used when threads within a warp have different values to be written in an array v, but some may be invalid due to previous filtering. This primitive reorganizes v and valid values are written at the beginning of v and invalid ones at the end of v.

The function receives the array v, the starting position in v where values should be written, the value itself, and a boolean indicating whether the value is valid. The threads call the *ballot_sync* warp exchange primitive to build a mask that gathers the *valid* value of all threads (line 2). Line 3 counts the 1's in the mask (*popc*), representing the number of valid values the warp will write. Each thread counts the amount of valid (line 4) and invalid (line 5) values that the threads with lower lane will write, and threads use this information to calculate the exact position the values will be written in v (lines 6 and 7), depending on whether they are valid or not. As invalid values are written after the valid ones, the size of the array is increased only by the number of valid values (line 8). The positions written by threads are contiguous to ensure coalesced memory requests, and all threads write their values to guarantee a divergence-free execution. This primitive is essential to allow the optimization that removes the compact phase of DuMato workflow, as it will be used to reorganize the extensions inside the filter phase.

Algorithm 7: Primitive *write*.

```
1 $void\ write(v, start, value, valid):$
2   $valids \leftarrow ballot\_sync(valid);$
3   $amount\_valids \leftarrow popc(valids);$
4   $valids\_offset \leftarrow count\_1\_right(valids, lane);$
5   $invalids\_offset \leftarrow amount\_valids + count\_0\_right(valids, lane);$
6   $pos \leftarrow start + (valid\ ?\ valids\_offset\ :\ invalids\_offset);$
7   $v[pos] \leftarrow value;$
8   $v.len\ += amount\_valids$
```

Warp-Centric Implementation of Enumeration Phases. Next we detail the warp-centric implementation of each subgraph enumeration phase depicted in Fig. 4. Algorithm 8 presents the implementation of the *Extend* phase, the BFS phase of DFS-wide that performs custom operations implemented by function pointer *op* (e.g., intersection) using the specified extensions, cache lines and adjacency lists provided by the masks. Lines 2-3 are initial SISD steps that check whether the extensions have already been generated and, in case they have not, the *eg* flag is set to true as the extensions will be generated. After initializing the current extensions (line 5), lines 7-19 iterate over the input masks to call the function *op* and perform the desired operations with the appropriate sets and the results are gradually stored in the current $TE.ext$.

Algorithm 8: *Extend* primitive.

```
1 $void\ extend(TE, m\_ext, m\_cache, m\_adj, op):$
2   $eg \leftarrow TE.eg[TE.len - 1];$
3   $if(!eg):$
4     $TE.eg[TE.len - 1] \leftarrow true;$
5     $TE.ext[TE.len - 1] \leftarrow \emptyset;$
6     $i \leftarrow 0;$
7     $while(i < k - 1):$
8       $bit \leftarrow lowest\_bit(m\_ext);$
9       $if(bit):$
10        $TE.ext[TE.len - 1] \leftarrow op(TE.ext[TE.len - 1], TE.ext[i]);$
11      $bit \leftarrow lowest\_bit(m\_cache);$
12      $if(bit):$
13        $TE.ext[TE.len - 1] \leftarrow op(TE.ext[TE.len - 1], TE.cache[i]);$
14      $bit \leftarrow lowest\_bit(m\_adj);$
15      $if(bit):$
16        $TE.ext[TE.len - 1] \leftarrow op(TE.ext[TE.len - 1], TE.adj[i]);$
17      $m\_ext \leftarrow m\_ext >> 1;$
18      $m\_cache \leftarrow m\_cache >> 1;$
19      $m\_adj \leftarrow m\_adj >> 1;$
```

Filter iterates over a set of extensions and removes those that do not fulfill property P. As shown in Algorithm 9, it receives the current traversal and a function pointer P, which returns a boolean to indicate whether an extension is valid. Each thread within the warp gets an extension (line 5) and passes it to the P function (line 6). For example, one of the filters used in clique counting checks whether the id of an extension is lower than the id of *tr*'s last vertex. P functions are warp-centric and can be implemented using DuMato primitives to access the TE data structure. Lines 7-8 write the extensions to the extensions array, keeping only the valid ones. The remodeling of this phase using the *write* primitive allowed the removal of the *compact* phase from the execution workflow, as this primitive writes values back to the extensions array keeping them in contiguous memory positions. Lines 9-10 compute the number of valid extensions after filtering.

Algorithm 9: Filter primitive.

```
1  void filter(TE, P, args)
2    e ← TE.ext[TE.len − 1];
3    produced ← 0;
4    for(i ← warp_lane; i < extensions.len ; i+ = warp_size) :
5      extension ← extensions[i];
6      valid ← P(TE, extension, args)
7      TE.ext[TE.len − 1].len − = warp_size
8      write(TE.ext[TE.len − 1], produced, valid, extension)
9      amount ← popc(ballot_sync(valid));
10     produced ← produced + amount;
```

Move implements the DFS phase of DFS-wide, thus allowing a warp to move forward or backward in the enumeration of a traversal (recursion step). It is primarily a synchronization step to update the current traversal to the warp. Thus, most of its steps are SISD.

Aggregate produces outputs of the GPM algorithms and two of those primitives (*aggregate_count* and *aggregate_store*) are standard. The first counts, for instance, the total of visited subgraphs matching a property, while the latter outputs visited subgraphs to the CPU asynchronously whenever it is required.

The most complex and challenging aggregate primitive for GPU implementation is the *aggregate_pattern* is used to output counters for each possible canonical representative that matches the desired property, such as in the motif counting. The complexity holds on the conversion of a visited traversal into its canonical representative (canonical relabeling). Due to that, All subgraph enumeration systems in the literature perform this operation on the CPU using tools such as Nauty [39]. We propose a novel pre-processed dictionary to allow this conversion on the GPU (Fig. 6).

We provide a pre-processed input dictionary that converts each possible induced traversal with k vertices from its raw bitmap representation (Fig. 6(a)) to its canonical representative with the compact bitmap representation (Fig. 6(c)). When a warp visits a traversal and the *aggregate_pattern* is called, the traversal is converted to the compact bitmap representation using the dictionary, and the corresponding number is used as the index for the local warp counters.

Local warp counters whose size is the number of canonical representatives are possible only due to our fast mechanism to convert a traversal to a compact bitmap representation using the pre-processed dictionary. These dictionaries can be used in any dataset and application that requires canonical relabeling (e.g., frequent subgraph mining [16] and subgraph matching [22]), and we provide them as input files for different k values. To the best of our knowledge, this is the first work to propose canonical relabeling on the GPU.

4.3. Warp-level load balancing

Our load-balancing mechanism decides when and how to perform the workload redistribution. It is implemented on the CPU and is depicted by the *Load Balancing* box in Fig. 4. This layer communicates asynchronously with the GPU by setting flags to indicate to the GPU's control phase when a device-wide synchronization step is required to perform a load-balancing step. This is implemented through the functions *when_rebalance* and *how_rebalance* depicted in Algorithm 10. Both functions receive a *DM_info* argument containing a copy of the main GPU data structures and control flags. In the *when_rebalance*, the warps' activity information is continuously read by the CPU (lines 2 and 10), and if the number of idle warps is found to be higher than a threshold (*thr*), the workload balancing is carried out (line 4). The GPU *lb* flag is set to true to inform warps that the execution should be interrupted, and this flag is read by the *Control* phase on GPU. The CPU then waits for the kernel (all warps) to finish and executes *how_rebalance* to perform *donations* between warps. Given two warps w_1 and w_2, a **donation** from w_1 to w_2 is the extraction of one active traversal from w_1's queue of jobs and its insertion into w_2's queue of jobs. We say w_1 is the **donator**. Once rebalancing is completed, line 9 restarts the execution.

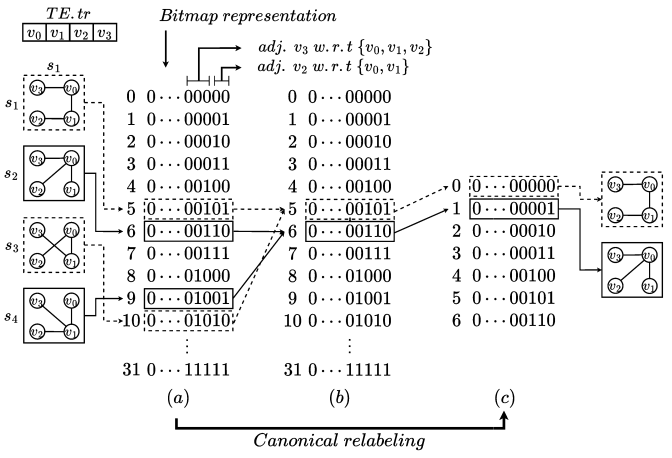


Fig. 6. Conversion of subgraphs from bitmap representation (a) to their canonical representatives (c).

Algorithm 10: CPU code for load balancing.

```
1  void when_rebalance(DM_info gpu){
2    flags ← gpu.read_flags();
3    while(flags.active_warps > 0):
4      if(flags.idle_warps > thr):
5        gpu.lb ← true;
6        gpu.waitKernel();
7        how_rebalance(gpu);
8        gpu.lb ← false;
9        gpu.runKernel();
10     flags ← gpu.read_flags();
11 void how_rebalance(DM_info gpu){
12   idles ← list(gpu.idles);
13   actives ← heap(gpu.actives);
14   total_weight ← sum_weight(actives);
15   avg_weight ← total_weight/|warps|;
16   for(i ← 0 ; i < donations ; i + +):
17     for each(idle ∈ idles):
18       donator ← actives.pop_heap();
19       idle.jobs.push(extract(donator));
20       if(donator.weight > avg_weight):
21         actives.push_heap(donator);
```

In our preliminary version of this work [18], the *donation* among warps (*how_rebalance*) did not consider the cost of the jobs (traversals) assigned to a warp and was only able to *donate* a single job among busy and idle warps. Here, we improved this strategy with a redistribution approach that selects several jobs from warp donators using information from their current traversal and extensions, rather than picking donators in a round-robin style.

The function *how_rebalance* depicts our new strategy. We create a list of idle warps (line 12) and a max heap with the active ones (line 13). The criterion used in the heap ordering is the warp weight, which is the sum of the size of its arrays of extensions ($TE.ext$). Once the total

Table 3
Characteristics of datasets used for evaluation.

Dataset	V(G)	E(G)	Avg. Deg.	Density	Max. Deg.
Citeseer [16]	3.2 K	4.5 K	2.77	8.51×10^{-4}	99
ca-AstroPh [33]	18.7 K	198.1 K	21.10	1.12×10^{-3}	504
Mico [16]	96.6 K	1.08 M	22.35	2.31×10^{-4}	1359
com-DBLP [50]	317 K	1.04 M	6.62	2.08×10^{-5}	343
soc-Pokec [50]	1.6 M	30.6 M	37.50	1.14×10^{-5}	14854
com-LiveJournal [50]	3.9 M	34.6 M	17.35	4.34×10^{-6}	14815

Table 4
Speedup as optimizations are activated in DuMato. Cells with "≥": only the speedup baseline exceeded 24 hours. Cells with "-": both variations exceeded 24 hours.

		Impl.	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
Clique	ca-Astr.	DFS → WC	22	62.1	100	83.8	86.1	97.5	≥57.6	≥10.2
		WC → WC_RRLB	0.1	0.5	1.3	4.4	14.2	28.3	44.4	61.2
		WC_RRLB → WC_JLB	0.9	0.9	0.9	1	1.1	1.1	1	1
	Mico	DFS → WC	12.7	57	76.3	≥12.1	-	-	-	-
		WC → WC_RRLB	0.8	2.5	9.2	17.1	≥7.6	-	-	-
		WC_RRLB → WC_JLB	0.9	1.6	1.5	1.1	1	-	-	-
	DBLP	DFS → WC	14	30.3	46.6	54.9	64.3	≥4.6	-	-
		WC → WC_RRLB	0.1	0.5	4.9	22.3	48	76.5	≥29.8	≥2.8
		WC_RRLB → WC_JLB	1	0.9	1.1	1.3	1.2	1.1	1	1
	LiveJr.	DFS → WC	14.8	132.6	≥54	-	-	-	-	-
		WC → WC_RRLB	2.4	1.4	2.5	≥9.2	-	-	-	-
		WC_RRLB → WC_JLB	1.7	1.3	2.8	1.2	-	-	-	-
	Pokec	DFS → WC	19.3	69.1	84.9	91.5	152.1	382	436.2	363.3
		WC → WC_RRLB	4.8	2.5	1.9	1.6	1.3	1.1	1.5	2.5
		WC_RRLB → WC_JLB	2.8	2.6	2.1	2	1.7	2	2.1	2.1
Motifs	Citeseer	DFS → WC	1	9	8.3	8.4	13.8	≥7.1	-	-
		WC → WC_RRLB	0.5	2	12.9	37	65.5	98.3	-	-
		WC_RRLB → WC_JLB	0.2	0.3	0.5	1.2	1.6	1.3	-	-
	ca-Astr.	DFS → WC	19.9	26.3	≥14.7	-	-	-	-	-
		WC → WC_RRLB	0.9	6.8	36.8	≥3.3	-	-	-	-
		WC_RRLB → WC_JLB	1.3	1.8	1.2	1	-	-	-	-
	Mico	DFS → WC	24	23.8	-	-	-	-	-	-
		WC → WC_RRLB	1.3	11.9	≥10.7	-	-	-	-	-
		WC_RRLB → WC_JLB	1.7	2.4	1.1	-	-	-	-	-
	DBLP	DFS → WC	14.9	20.3	19.7	-	-	-	-	-
		WC → WC_RRLB	0.4	3.6	21.3	≥27.6	-	-	-	-
		WC_RRLB → WC_JLB	1.3	2.5	2.2	1.2	-	-	-	-

weight of active warps is computed, we calculate the average weight (line 15), which will be used as a threshold (line 20) to decide whether an active warp will donate extensions. Warps carry a list of traversals to be processed, called *jobs*. Given an idle warp, we pop the active warp with the highest weight (line 18), get one of its extensions, and push it to the list of jobs of the idle warp. If the weight of the *donator* warp is still higher than the average, it is pushed back to the heap (lines 20-21). This strategy improves the previous round-robin approach, reducing the number of calls to the load-balancing layer as the warps take longer to get idle after balancing.

5. Experimental evaluation

This section evaluates the gains of each optimization proposed by DuMato and compares it to the state-of-the-art GPM systems. To illustrate, we use three GPM algorithms: *clique counting*, *motif counting* and *subgraph matching*, presented in Section 4.2.2. Table 3 presents the datasets employed. CPU experiments were conducted on a machine with an Intel Xeon Silver 4108 CPU (16 threads with hyperthreading), 48 GB of RAM, and Ubuntu 18.04. GPU experiments used an NVIDIA TITAN V with 12 GB and CUDA 10.1. The time limit adopted for each execution was 24 hours. Every execution was run three times and demonstrated low variability (standard deviations in 0.06%-1.07%). Results for LiveJournal and Pokec for the motif counting application are not presented because it exceeds the 24-hour limit even for small subgraph sizes ($k > 4$). Results for the clique counting application using the Citeseer dataset are not presented because they are not representative, as this dataset contains a small number of cliques and all systems enumerate them in a few milliseconds.

We have evaluated four versions (implementations) of DuMato that vary according to the optimizations leveraged: DM_DFS in which each GPU thread receives a traversal and calculates the enumeration using the DFS; DM_WC in which GPU warps receives traversals for processing and uses the DFS-wide traversal and the warp-centric workflow; DM_WC_RRLB that includes load balancing and is the fastest version from our previous work [18]; DM_WC_JLB that uses our new load-balancing strategy (Section 4.3). The speedups as optimizations are activated in DuMato are presented in Table 4 for the *motif counting* and *clique counting* algorithms. We have varied the number of threads used and balancing threshold to choose these parameters and the results shown that a value of 102,400 threads and a threshold of 30% led to the best performance for most of our case. Details on this evaluation are available in Section S1. As such, we have used these values for the rest of the experiments.

5.1. Gains due to optimizations

The DM_DFS assigns traversals per thread that process them independently. As it may result in a different execution path, threads within a warp will diverge through the enumeration, deteriorating warp and memory efficiency. Divergences are reduced and memory access pattern is improved by the *DM_WC* version, which attains speedups up to two orders of magnitude (e.e., Clique, LiveJournal and $k = 4$) w.r.t. the *DM_DFS*.

We have executed *DM_DFS* and *DM_WC* versions with the CUDA NVProf profiling tool [10], which allowed us to measure the impacts of our optimizations at the hardware level. The profiling results for the DBLP dataset and k up to 4 are shown in Table 5. Results for the other

Table 5
DM_DFS vs DM_WC: execution and memory metrics.

App.	k	Memory (load transactions)			Execution (inst. per warp)		
		DM_DFS	DM_WC	Improvement	DM_DFS	DM_WC	Improvement
Clique	3	618.1 M	212.7 M	2.9×	3.3 M	876.6 K	3.8×
	4	6.7 B	852.4 M	7.9×	50.5 M	5.1 M	9.9×
Motifs	3	3.3 B	597.0 M	5.53×	17.5 M	2.6 M	7.36×
	4	134.7 B	22.8 B	5.90×	1.9 B	143.2 M	13.3×

datasets are similar. Two categories of metrics (execution and memory) allow us to quantify DuMato's effectiveness in using GPU's massive parallelism, execution, and memory hierarchy. The metric *inst_per_warp* indicates the average number of instructions warps need to execute the respective kernel. Our goal with the improvements in regularity is to allow a lockstep execution within a warp, reducing the number of total instructions issued. The metrics *gld_transactions* quantifies the total amount of global memory transactions needed to service read requests. Our optimizations improve the memory access pattern, providing more coalesced requests and reducing the number of transactions.

The execution efficiency of our warp-centric implementation is confirmed by the reduction in the total number of instructions per warp needed by the *DM_WC* version, with improvements ranging from 3.8x and 13.3x. These optimizations increased the amount of coalesced memory requests, thus reducing the total amount of memory transactions from 2.9× to 7.9×.

The DM_WC_RRLB provided significant performance gains vs. DM_WC, reaching speedups up to 98× (Motifs, Citeseer and k = 8). The impact of load-balancing is more effective as k increases. This happens because the load imbalance is higher in these cases due to the skewness of real-world datasets. For more skewed datasets (Citeseer, ca-AstroPh, and DBLP), the workload distribution becomes imbalanced more quickly and our load-balancing strategy therefore attains higher gains.

The DM_WC_JLB that donates multiple jobs per warp was first tuned to set its number of donations to 16 (See Section S2 for details). As shown in Table 4, it improves the DM_WC_RRLB performance in all cases. More details on load balancing evaluation are available in Section S3.

5.2. Comparison to other GPM systems

This section compares the best performing version of DuMato/DMT (version including all optimizations: DM_WC_JLB) to the state-of-the-art subgraph enumeration systems for GPU: G2Miner/G2M [7] (pattern-aware and DFS), RPS/RPS [21] (pattern-aware and DFS) and Pangolin/PAN [9] (pattern-oblivious and BFS). We also compared DuMato to the CPU systems Peregrine/PER [27] (pattern-aware) and Fractal/FRA [13] (pattern-oblivious). All these systems represent different architectures, traversal strategies (BFS/DFS), and enumeration paradigms (pattern-oblivious/pattern-aware).

The experimental results for the motif counting and clique counting algorithms are presented in Table 6. We can observe that DuMato can enumerate larger subgraphs than any other system. This is possible due to the DFS-wide traversal strategy, which reduces the memory consumption and mitigates the impacts of combinatorial explosion, thus allowing the visitation of larger subgraphs using an affordable amount of memory; the efficient load-balancing layer, which mitigates the load imbalance as we increase k and reduces the impacts of data skewness; the flexibility in the choice of the enumeration paradigm, as the pattern-oblivious allowed the visitation of several patterns in parallel up to $k = 8$ (motif counting) without the need of thousands of exploration plans, and the pattern-aware provided an optimized version of the application relying on a single pattern (clique counting). Although Table 6 showed only subgraph sizes up 14, DuMato was able to reach subgraph sizes up to 28 (Clique counting, *Pokec* dataset) in less than a minute. To the best of our knowledge, subgraphs of such size have not been explored by any other enumeration system searching for exact outputs, demonstrating our scalability and memory awareness.

Pangolin uses BFS, which materializes all the intermediate enumeration states and facilitates regular execution and load balancing, providing good execution times for small enumerated subgraphs. However, as the size of the enumerated subgraphs increases, it crashes due to its high-memory consumption caused by the combinatorial explosion of intermediate enumeration states. Similarly to DuMato, Fractal uses DFS and the pattern-oblivious enumeration paradigm, but it executes on CPUs. DuMato attained consistent speedups w.r.t. Fractal in all executions.

Peregrine is a pattern-aware DFS system, and we attained speedups up to 105× when larger patterns are enumerated (motif application, Citeseer dataset, and $k = 8$). As we increase the size of the enumerated subgraphs in the motif counting application, we also increase the number of valid patterns. This increase does not generate pre-processing overheads to DuMato, as the pattern-oblivious strategy does not use the patterns to guide the exploration. On the other hand, Peregrine relies on exploration plans for each pattern, and this growth in the number of patterns also increases its pre-processing costs to generate the exploration plans. Additionally, as the motif counting searches for all possible subgraphs, some of Peregrine's exploration plans may not generate valid subgraphs, wasting computational resources. Peregrine's best-case scenario is the enumeration of one specific pattern, and even in this scenario (clique) we attained consistent speedups.

The reduction in the memory consumption of parallel subgraph enumeration in the exploration of larger subgraphs, along with the efficient strategies for regular parallel processing, confirm our hypotheses that subgraph enumeration on GPU must deal with irregularity and combinatorial explosion in order to design and implement GPM algorithms efficiently on this architecture. Our efficient pattern-oblivious design also allows the exploration of more patterns in parallel compared to the state-of-the-art GPM systems.

For the comparison with the state-of-the-art frameworks we also executed the subgraph matching application (Table 7). The patterns used were quasi-cliques such that q_i is a graph with i vertices and $(i \times (i-1)/2) - 1$ edges. The exploration plan used was extracted from the heuristic proposed by Fractal [13]. G2Miner, Pangolin, and RPS could not generate functional GPU kernels to execute for any of these patterns. Thus, their results are not presented. G2Miner provides a code generator to create new GPU kernels for specific patterns, but it does not generate functional GPU code for new patterns. Pangolin is a pattern-oblivious system that does not provide an implementation for subgraph matching. In order to match a new pattern, RPS needs the set of symmetry-breaking rules needed for the pattern. However, the symmetry-breaking rules needed to break automorphism depend on the exploration plan, and none of the symmetry-breaking rules we provided could execute RPS properly. DuMato presents significant speedups, achieving higher speedups as we increase the size of the patterns. For example, there is a small increase in the execution time of DuMato from q_9 to q_{10} in the *Pokec* dataset, while the other systems are more impacted, showing the scalability of our strategies as we increase k.

6. Conclusion and future work

In this work, we propose novel strategies to mitigate the main challenges for efficient subgraph enumeration on GPUs: irregularity, which

Table 6
Execution time (seconds) of DuMato and baselines (GPU and CPU). Cells with "-": exceeded 24 hours. "ERR": errors during execution. "NS": not supported. "OOM": out-of-memory. "INC": finished with incomplete results.

		$k=$	3	4	5	6	7	8	9	10	11	12	13	14
Motifs	Citeseer	DMT	0.11	0.12	0.24	**0.67**	**5.06**	**96.95**	-	-	-	-	-	-
		G2M	**0.01**	ERR	NS	NS	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	**0.01**	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	**0.01**	**0.01**	**0.05**	3.47	537.66	-	-	NS	NS	NS	NS	NS
		FRA	5.17	5.20	5.69	12.44	163.48	-	-	-	-	-	-	-
	ca-Astroph	**DMT**	0.25	1.47	**126.78**	**23.62 K**	-	-	-	-	-	-	-	-
		G2M	**0.01**	**0.05**	NS	NS	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	0.21	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	**0.01**	0.57	132.90	52.80 K	-	-	-	-	-	-	-	-
		FRA	9.13	435.64	4.72 K	-	-	-	-	-	-	-	-	-
	Mico	**DMT**	0.47	23.27	**7.62 K**	-	-	-	-	-	-	-	-	-
		G2M	**0.01**	**0.84**	NS	NS	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	3.31	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	0.06	6.57	7.92 K	-	-	-	-	-	-	-	-	-
		FRA	16.43	474.46	-	-	-	-	-	-	-	-	-	-
	DBLP	**DMT**	0.13	1.11	**31.90**	**2.64 K**	-	-	-	-	-	-	-	-
		G2M	**0.01**	0.84	NS	NS	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	**0.17**	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	0.07	0.95	78.59	50.95 K	-	-	-	-	-	-	-	-
		FRA	14.33	37.62	1.43 K	-	-	-	-	-	-	-	-	-
Clique	ca-Astroph	**DMT**	0.13	0.15	0.43	0.86	2.19	7.82	**32.48**	**137.04**	**561.12**	**2.13 K**	**7.44 K**	**24.57 K**
		G2M	**0.01**	**0.01**	**0.01**	**0.08**	0.68	**5.01**	NS	NS	NS	NS	NS	NS
		RPS	0.02	0.04	0.17	1.85	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	**0.01**	0.02	0.11	**0.61**	OOM	OOM	OOM	OOM	-	-	-
		PER	**0.01**	0.10	0.83	6.38	43.56	272.42	1.55 K	7.93 K	36.26 K	-	-	-
		FRA	8.17	9.75	15.89	78.09	439.16	2.30 K	12.89 K	57.02 K	-	-	-	-
	Mico	**DMT**	0.31	1.08	13.93	373.14	10.91 K	-	-	-	-	-	-	-
		G2M	**0.01**	**0.02**	**0.74**	**31.98**	**1.16 K**	**39.58 K**	NS	NS	NS	NS	NS	NS
		RPS	0.11	0.37	14.59	953.00	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	0.05	2.93	OOM	-	-	-	-	-	-	-	-
		PER	0.09	1.81	82.67	3.66 K	-	-	-	-	-	-	-	-
		FRA	14.17	48.53	1.44 K	56.72 K	-	-	-	-	-	-	-	-
	DBLP	**DMT**	0.13	0.29	0.47	2.09	19.49	229.52	**2.77 K**	**29.90 K**	-	-	-	-
		G2M	**0.01**	**0.01**	**0.02**	**0.37**	**8.16**	**148.23**	NS	NS	NS	NS	NS	NS
		RPS	0.04	0.07	0.24	3.62	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	0.01	0.03	0.50	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	0.11	0.16	1.36	25.92	531.88	9.35 K	-	-	-	-	-	-
		FRA	13.44	14.32	22.72	186.97	2.52 K	35.51 K	-	-	-	-	-	-
	LiveJournal	DMT	4.70	22.91	232.53	8.00 K	-	-	-	-	-	-	-	-
		G2M	0.02	**0.21**	**6.39**	**318.98**	**14.95 K**	-	NS	NS	NS	NS	NS	NS
		RPS	2.16	6.18	154.78	9.63 K	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	**0.01**	0.53	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		PER	3.91	26.66	1.06 K	64.74 K	-	-	-	-	-	-	-	-
		FRA	394.85	901.05	16.06 K	-	-	-	-	-	-	-	-	-
	Pokec	DMT	1.44	2.91	4.97	6.32	8.84	9.32	**11.32**	**13.03**	**17.45**	**17.88**	**19.56**	**23.38**
		G2M	**0.01**	**0.03**	**0.06**	**0.10**	**0.18**	**0.39**	NS	NS	NS	NS	NS	NS
		RPS	1.79	3.16	4.44	5.97	NS	NS	NS	NS	NS	NS	NS	NS
		PAN	0.06	0.21	0.30	0.39	0.52	ERR	ERR	ERR	ERR	ERR	ERR	ERR
		PER	2.46	6.93	14.19	24.61	39.93	61.94	93.85	144.47	221.63	344.19	546.65	838.45
		FRA	172.41	212.27	293.00	495.62	577.57	812.94	1.03 K	1.52 K	1.70 K	1.96 K	2.26 K	2.42 K

limits the use of GPU's massive parallelism and HBRAM; combinatorial explosion, which creates high memory demands and limits the scalability of GPM algorithms; flexibility in the enumeration paradigm, as the tradeoff between using these paradigms is hard to quantify and current GPU solutions support either pattern-oblivious or pattern-aware. Our DFS-wide traversal strategy provides a good tradeoff between memory locality and low memory consumption for the intermediate enumeration states, thus improving the efficiency in accessing GPU's HBRAM and reducing the impacts of combinatorial explosion.

Our warp-centric enumeration workflow uses the DFS-wide data structures to implement subgraph enumeration through efficient SIMD/SISD lockstep phases, reducing divergences and improving GPU's HBRAM efficiency through memory coalescence. In this work, we refactored the enumeration phases using three efficient warp-centric core primitives (*find_one*, *find_many*, and *write*). This not only improved the system's comprehensiveness but also allowed the removal of the compaction phase from the enumeration workflow.

Our load-balancing strategies mitigate the imbalance caused by the irregular processing during the parallel subgraph enumeration. We proposed a lightweight warp-level layer performed by the CPU, which monitors GPU occupancy to rebalance when utilization is low. Two custom functions must be provided to this layer: *when_rebalance*, which uses a threshold to infer when GPU is idle and a workload redistribution is necessary; *how_rebalance*, which improves our previous strategy to redistribute enumeration jobs considering the weight of each warp. Jobs are now extracted from the heaviest warps, thus allowing a better load balancing than our previous round-robin strategy. Furthermore, warps receive several jobs in a rebalancing, increasing the GPU occupancy and reducing the calls to the load-balancing layer.

Our general-purpose subgraph enumeration system (DuMato) uses our novel strategies to allow efficient implementations of GPM algorithms on GPUs using high-level primitives and an adaptive execution workflow. To the best of our knowledge, DuMato is the first system to support both enumeration paradigms (pattern-oblivious/pattern-aware), allowing the users to understand and exploit the best-case scenario of each paradigm and achieve better performance results.

In the future, we plan to improve the efficiency of the warp-centric enumeration phases by splitting physical warps into virtual ones (sub-

Table 7
Execution time (seconds) of DuMato and baselines (GPU and CPU) for the subgraph matching application.

	System	*q*4	*q*5	*q*6	*q*7	*q*8	*q*9	*q*10
Citeseer	**DuMato**	0.11	0.13	0.13	0.14	∅	∅	∅
	Peregrine	0.01	0.01	0.01	0.19	∅	∅	∅
	Fractal	1.78	1.84	1.78	2.05	∅	∅	∅
ca-Astr.	**DuMato**	0.15	0.46	1.59	6.48	34.04	181.45	924.89
	Peregrine	0.02	0.30	2.78	23.04	165.11	1.23 K	16.83 K
	Fractal	5.05	17.25	140.05	1.26 K	8.35 K	62.10 K	–
Mico	**DuMato**	0.74	7.48	234.49	9.64 K	–	–	–
	Peregrine	0.15	5.85	321.27	14.71 K	–	–	–
	Fractal	39.59	2.80 K	–	–	–	–	–
DBLP	**DuMato**	0.15	0.50	3.90	53.56	846.29	12.07 K	–
	Peregrine	0.10	0.34	3.44	73.22	1.54 K	26.85 K	–
	Fractal	9.52	35.58	759.04	18.28 K	–	–	–
LiveJr.	**DuMato**	10.36	172.09	4.22 K	–	–	–	–
	Peregrine	5.63	76.00	3.32 K	–	–	–	–
	Fractal	672.82	39.58 K	–	–	–	–	–
Pokec	**DuMato**	2.21	9.94	14.09	17.34	19.50	21.64	27.67
	Peregrine	3.30	10.28	20.55	35.11	57.80	281.19	11.38 K
	Fractal	182.09	242.56	371.92	495.40	749.39	1.18 K	1.61 K

warps). This will enable us to modulate subwarps sizes dynamically according to the sizes of neighborhoods, which should improve warp execution efficiency when visiting the adjacency lists while generating extensions. We also plan extending our system with a multi-GPU version to accelerate it further. Another promising direction for future work refers to the investigation of novel approaches for load balancing in the domain, given the high impact of this optimization to the performance. As such, we expect to compare our approaches to those proposed in the STMatch [48] that, differently from DuMato, performs stealing within and across threadbloks inside the GPU. The STMatch strategies could also benefit from our weighted task redistribution, leading to novel approaches.

CRediT authorship contribution statement

Samuel Ferraz: Conceptualization, Data curation, Investigation, Methodology, Project administration, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Vinicius Dias:** Conceptualization, Formal analysis, Methodology, Writing – original draft, Writing – review & editing. **Carlos H.C. Teixeira:** Conceptualization. **Srinivasan Parthasarathy:** Conceptualization, Formal analysis, Writing – original draft. **George Teodoro:** Conceptualization, Methodology, Supervision, Writing – original draft, Writing – review & editing. **Wagner Meira:** Conceptualization, Funding acquisition, Methodology, Project administration, Resources, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This study was partially funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CAPES/CNPq (grants 407108/2022-0 and 313017/2022-0), FAPEMIG (grants APQ-00161-20 and PPE-00030-21), CNPq-AWS (grants 400070/2023-5 and 440088/2020-8), and PROCAD/UFMG. Srinivasan Parthasarathy would like to acknowledge support from National Science Foundation (USA) under grants CNS-2112471 and OAC-2018627. Any opinions, findings, and conclusions in this material are those of the author(s) and may not reflect the views of the respective funding agencies.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jpdc.2024.104903.

References

[1] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, F. Jamour, Scalemine: scalable parallel frequent subgraph mining in a single large graph, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 61:1–61:12, http://dl.acm.org/citation.cfm?id=3014904.3014986.
[2] M. Almasri, I.E. Hajj, R. Nagi, J. Xiong, W.-m. Hwu, Parallel k-clique counting on gpus, in: Proceedings of the 36th ACM International Conference on Supercomputing, ICS '22, Association for Computing Machinery, New York, NY, USA, 2022.
[3] F. Bi, L. Chang, X. Lin, L. Qin, W. Zhang, Efficient subgraph matching by postponing cartesian products, in: SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1199–1214.
[4] B. Bringmann, S. Nijssen, What is frequent in a single graph?, in: T. Washio, E. Suzuki, K.M. Ting, A. Inokuchi (Eds.), Advances in Knowledge Discovery and Data Mining, Springer, Berlin, Heidelberg, 2008, pp. 858–863.
[5] G. Buehrer, S. Parthasarathy, Y. Chen, Adaptive parallel graph mining for CMP architectures, in: Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China, IEEE Computer Society, 2006, pp. 97–106.
[6] G. Buehrer, S. Parthasarathy, M. Goyder, Data mining on the cell broadband engine, in: P. Zhou (Ed.), Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008, ACM, 2008, pp. 26–35.
[7] X. Chen, Arvind, efficient and scalable graph pattern mining on GPUs, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), USENIX Association, Carlsbad, CA, 2022, pp. 857–877, https://www.usenix.org/conference/osdi22/presentation/chen.
[8] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, J. Cheng G-miner, An efficient task-oriented graph mining system, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, Association for Computing Machinery, New York, NY, USA, 2018.
[9] X. Chen, R. Dathathri, G. Gill, K. Pingali, Pangolin: an efficient and flexible graph mining system on cpu and gpu, Proc. VLDB Endow. (2020).
[10] N. Corporation, Toolkit documentation, https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2022.
[11] M. Danisch, O. Balalau, M. Sozio, Listing k-cliques in sparse real-world graphs*, in: Proceedings of the 2018 World Wide Web Conference, WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 2018, pp. 589–598.
[12] M. Danisch, O. Balalau, M. Sozio, Listing k-cliques in sparse real-world graphs, in: WWW '18, 2018.
[13] V. Dias, C.H.C. Teixeira, D. Guedes, W. Meira, S. Parthasarathy, Fractal: a general-purpose graph pattern mining system, in: SIGMOD '19, 2019.
[14] V. Dodeja, M. Almasri, R. Nagi, J. Xiong, W. mei Hwu, PARSEC: PARallel subgraph enumeration in CUDA, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2022.
[15] A. Duma, A. Topirceanu, A network motif based approach for classifying online social networks, in: SACI '14, 2014.
[16] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, P. Kalnis, Grami: frequent subgraph and pattern mining in a single large graph, Proc. VLDB Endow. (2014).

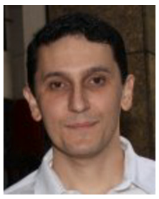
[17] D. Eppstein, M. Löffler, D. Strash, Listing all maximal cliques in large sparse real-world graphs, ACM J. Exp. Algorithmics 18 (Nov. 2013), https://doi.org/10.1145/2543629.
[18] S. Ferraz, V. Dias, C.C. Teixeira, G. Teodoro, W. Meira, Efficient strategies for graph pattern mining algorithms on gpus, in: 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE Computer Society, Los Alamitos, CA, USA, 2022, pp. 110–119, https://doi.ieeecomputersociety.org/10.1109/SBAC-PAD55451.2022.00022.
[19] C. Gui, X. Liao, L. Zheng, P. Yao, Q. Wang, H. Jin, Sumpa: efficient pattern-centric graph mining with pattern abstraction, in: PACT '21, 2021, pp. 318–330.
[20] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, K.-L. Tan, Gpu-accelerated subgraph enumeration on partitioned graphs, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1067–1082.
[21] W. Guo, Y. Li, K. Tan, Exploiting reuse for gpu subgraph enumeration, IEEE Trans. Knowl. Data Eng. 34 (09) (2020) 4231–4244, https://doi.org/10.1109/TKDE.2020.3035564.
[22] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, K.-L. Tan, Gpu-accelerated subgraph enumeration on partitioned graphs, in: SIGMOD, 2020.
[23] W.-S. Han, J. Lee, J.-H. Lee Turboiso, Towards ultrafast and robust subgraph isomorphism search in large graph databases, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 337–348.
[24] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating cuda graph algorithms at maximum warp, in: PPoPP '11, 2011.
[25] B. Hooi, K. Shin, H. Lamba, C. Faloutsos, Telltail: fast scoring and detection of dense subgraphs, in: AAAI '20, 2020.
[26] Y. Hu, H. Liu, H.H. Huang, Tricore: parallel triangle counting on gpus, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, 2018.
[27] K. Jamshidi, R. Mahadasa, K. Vora, Peregrine: a pattern-aware graph mining system, in: EuroSys '20, 2020.
[28] G. Jiang, Q. Zhou, T. Jin, B. Li, Y. Zhao, Y. Li, J. Cheng, Vsgm: view-based gpu-accelerated subgraph matching on large graphs, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22, IEEE Press, 2022.
[29] T. Junttila, P. Kaski, Engineering an efficient canonical labeling tool for large and sparse graphs, in: D. Applegate, G.S. Brodal, D. Panario, R. Sedgewick (Eds.), Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, SIAM, 2007, pp. 135–149.
[30] R. Kessl, N. Talukder, P. Anchuri, M. Zaki, Parallel graph mining with gpus, in: W. Fan, A. Bifet, Q. Yang, P.S. Yu (Eds.), Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, in: Proceedings of Machine Learning Research, vol. 36, PMLR, New York, New York, USA, 2014, pp. 1–16, https://proceedings.mlr.press/v36/kessl14.html.
[31] H. Kim, J. Lee, S.S. Bhowmick, W.-S. Han, J. Lee, S. Ko, M.H. Jarrah, Dualsim: parallel subgraph enumeration in a massive graph on a single machine, in: SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1231–1245.
[32] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, S. Yang, Scalable distributed subgraph enumeration, Proc. VLDB Endow. 10 (3) (2016) 217–228, https://doi.org/10.14778/3021924.3021937.
[33] J. Leskovec, J. Kleinberg, C. Faloutsos, Graph evolution: densification and shrinking diameters, ACM Trans. Knowl. Discov. Data (2007).
[34] W. Lin, X. Xiao, X. Xie, X. Li, Network motif discovery: a gpu approach, in: ICDE '15, 2015.
[35] G. Liu, L. Wong, Effective pruning techniques for mining quasi-cliques, in: ECMLP-KDD '08, 2008.
[36] Z. Lu, J. Wahlström, Community detection in complex networks via clique conductance, Sci. Rep. 8 (04) (2018), https://doi.org/10.1038/s41598-018-23932-z.
[37] D. Mawhirter, B. Wu, Automine: harmonizing high-level abstraction and high performance for graph mining, in: SOSP '19, 2019.
[38] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, B. Wu, Graphzero: a high-performance subgraph matching system, SIGOPS Oper. Syst. Rev. (2021).
[39] B.D. McKay, A. Piperno, Practical graph isomorphism, ii, J. Symb. Comput. (2014).
[40] P. Ribeiro, F. Silva, G-tries: a data structure for storing and finding subgraphs, Data Min. Knowl. Discov. 28 (2) (2014) 337–377, https://doi.org/10.1007/s10618-013-0303-4.
[41] R.A. Rossi, N.K. Ahmed, The network data repository with interactive graph analytics and visualization, in: AAAI, 2015, https://networkrepository.com.
[42] T. Shi, M. Zhai, Y. Xu, J. Zhai Graphpi, High performance graph pattern matching through effective redundancy elimination, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, 2020.
[43] O. Sporns, R. Kötter, Motifs in brain networks, PLoS Biol. (2004).
[44] S. Sun, Y. Che, L. Wang, Q. Luo, Efficient parallel subgraph enumeration on a single machine, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019.
[45] C.H.C. Teixeira, A.J. Fonseca, M. Serafini, G. Siganos, M.J. Zaki, A. Aboulnaga, Arabesque: a system for distributed graph mining, in: SOSP '15, 2015.
[46] H.-N. Tran, J.-j. Kim, B. He, Fast subgraph matching on large graphs using graphics processors, in: M. Renz, C. Shahabi, X. Zhou, M.A. Cheema (Eds.), Database Systems for Advanced Applications, Springer International Publishing, Cham, 2015, pp. 299–315.
[47] K. Wang, Z. Zuo, J. Thorpe, T.Q. Nguyen, G.H. Xu, Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine, in: OSDI'18, 2018.
[48] Y. Wei, P. Jiang, Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations, in: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, 2022, pp. 1–13.
[49] D. Yan, G. Guo, M.M. Rahman Chowdhury, M. Tamer Özsu, W.-S. Ku, J.C.S. Lui, G-thinker: a distributed framework for mining subgraphs in a big graph, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), 2020, pp. 1369–1380.
[50] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, in: MDS '12, 2012.
[51] X. Yang, S. Parthasarathy, P. Sadayappan, Fast sparse matrix-vector multiplication on gpus: implications for graph mining, Proc. VLDB Endow. 4 (4) (2011) 231–242.
[52] C. Zhao, Z. Zhang, P. Xu, T. Zheng, J. Guo, Kaleido: an efficient out-of-core graph mining system on a single machine, in: ICDE '20, 2020, pp. 673–684.

Samuel Ferraz received his Ph.D. in Computer Science from the Universidade Federal de Minas Gerais (UFMG), Brazil, in 2023. He is an Associate Professor of the Computer Science Department at Universidade Federal de Mato Grosso do Sul (UFMS), Brazil. His research interests include high-performance computing on GPUs and graph mining.

Vinícius Dias is an Associate Professor of the Computer Science Department at Universidade Federal de Lavras (UFLA), Brazil. He received his Ph.D. in Computer Science from the Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, in 2023. He also holds a M.Sc. in Computer Science from UFMG, in 2016, and a Bachelor degree from the Universidade Federal de Uberlândia, Brazil, in 2013. His research interests include performance of parallel and distributed systems, data mining, graph mining and machine learning with graphs.

Carlos Teixeira received his Ph.D. from the Computer Science Department at Universidade Federal de Minas Gerais (UFMG), Brazil, in 2022. He also earned both his bachelor's degree (2009) and master's degree (2011) in Computer Science from the same institution. He was a research associate at Qatar Computing Research Institute (2015) and visiting scholar at The Ohio State University (2009) and Purdue University (2016). His research is focused on data mining algorithms, machine learning, sampling methods, and data science.

Srinivasan Parthasarathy received the PhD degree from the Department of Computer Science, University of Rochester, Rochester, NY, in 1999. He is currently a professor with the Computer Science and Engineering Department, and the Biomedical Informatics Department, The Ohio State University. His research interests include high performance data analytics, graph analytics and network science, and machine learning and database systems. He is a Fellow of the IEEE, the AAIA, the Risk Institute and a Distinguished Fellow of the Robert Bosch Center for Data Science and AI.



George Teodoro received his M.Sc. and Ph.D. degrees in Computer Science from the Universidade Federal de Minas Gerais (UFMG), Brazil, in 2006 and 2010. He is professor at the Department of Computer Science at Universidade Federal de Minas Gerais. His primary areas of expertise include high performance runtime systems for efficient execution of biomedical, data-mining, and multimedia applications on distributed heterogeneous environments.

Wagner Meira Jr. obtained his Ph.D. from the University of Rochester, NY, in 1997 and is currently Professor of Computer Science at Universidade Federal de Minas Gerais, Brazil. He has published more than 300 papers in top venues and is co-author of the books Data Mining and Analysis — Fundamental Concepts and Algorithms, and Data Mining and Machine Learning – Fundamental Concepts and Algorithms, both published by Cambridge University Press in 2014 and 2020, respectively. His research focuses on scalability and efficiency of large scale parallel and distributed systems, from massively parallel to Internet-based platforms, and on data mining algorithms, their parallelization, and application to areas such as information retrieval, bioinformatics, cybersecurity, and health.