2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 978-1-6654-5155-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/SBAC-PAD55451.2022.00022

Efficient Strategies for Graph Pattern Mining Algorithms on GPUs

Samuel Ferraz*†, Vinicius Dias*‡, Carlos H. C. Teixeira*, George Teodoro*, Wagner Meira Jr.*
* *Computer Science Department, Federal University of Minas Gerais (UFMG) — Belo Horizonte, Brazil*
Email: {samuel.ferraz, viniciusvdias, carlos, george, meira}@dcc.ufmg.br
† *School of Computing, Federal University of Mato Grosso do Sul (UFMS) — Campo Grande, Brazil*
Email: samuel.ferraz@ufms.br
‡ *Department of Computing and Systems, Federal University of Ouro Preto (UFOP) — João Monlevade, Brazil*
Email: viniciusvdias@ufop.edu.br

***Abstract*—Graph Pattern Mining (GPM) is an important, rapidly evolving, and computation demanding area. GPM computation relies on subgraph enumeration, which consists in extracting subgraphs that match a given property from an input graph. Graphics Processing Units (GPUs) have been an effective platform to accelerate applications in many areas. However, the irregularity of subgraph enumeration makes it challenging for efficient execution on GPU due to typical uncoalesced memory access, divergence, and load imbalance. Unfortunately, these aspects have not been fully addressed in previous work. Thus, this work proposes novel strategies to design and implement subgraph enumeration efficiently on GPU. We support a depth-first search style search (DFS-wide) that maximizes memory performance while providing enough parallelism to be exploited by the GPU, along with a warp-centric design that minimizes execution divergence and improves utilization of the computing capabilities. We also propose a low-cost load balancing layer to avoid idleness and redistribute work among thread warps in a GPU. Our strategies have been deployed in a system named DuMato, which provides a simple programming interface to allow efficient implementation of GPM algorithms. Our evaluation has shown that DuMato is often an order of magnitude faster than state-of-the-art GPM systems and can mine larger subgraphs (up to 12 vertices).**

***Index Terms*—graph pattern mining, gpu, regular processing, load balancing**

I. Introduction

Graph pattern mining (GPM) aims to unveal relevant subgraph patterns in graphs, being widely used in different domains and applications from social media [1] to biological networks analysis [2]. It relies on subgraph enumeration over an input graph, which consists of visiting subgraphs that match a desired graph property. Subgraph enumeration incurs in high computational cost and memory demands as the size of the mined subgraphs increases [3]–[6]. For example, the small biological dataset *bio-diseasome* [1] (516 vertices, 1.2K edges) contains 112B induced subgraphs with 10 vertices, which would require around 4 TB of memory with a 4-byte integer per vertex to store all subgraphs.

A core operation in subgraph enumeration is the *subgraph extension*, in which subgraphs with $k+1$ vertices are obtained by the combination of a subgraph s with k vertices with a set of extensions (vertex ids) derived from the adjacency of vertices in s. Figure 1 illustrates this process using induced subgraphs s_1 and s_2. Subgraph extension of s_1 generates four extended subgraphs while s_2, six. The huge amount of subgraphs to be explored may cause long execution times, leading the pursuit of massively parallel architectures. Parallel

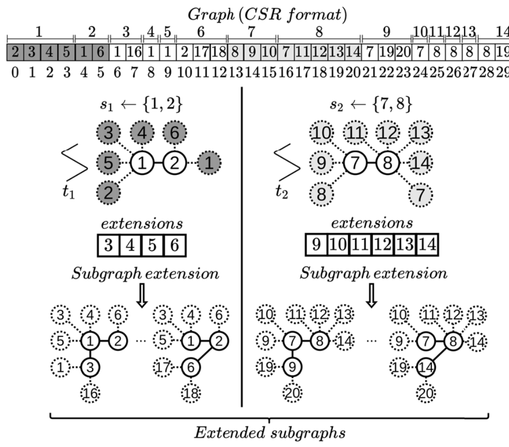


Fig. 1: Subgraph-centric parallel processing of subgraphs.

implementations of specific GPM algorithms were proposed for shared and distributed memory machines [7]–[10]. These solutions usually relax the definition of subgraph enumeration for each algorithm, reducing the computational cost and memory demands. Although they have attained good performance on some settings, the implementation of new algorithms was still a challenging and laborious process, as the restrictions used to design one GPM algorithm may not apply to others. This motivated the development of GPM systems, which support subgraph enumeration and allow the implementation of specific GPM algorithms by supporting custom graph properties. GPM systems offer a good tradeoff between programmability and performance [5], [6], [11]–[13].

Graphics Processing Units (GPUs) are successful in accelerating applications in many domains. However, the parallelization strategies for subgraph enumeration on GPU present

[1]https://networkrepository.com

major limitations concerning efficient use of GPU architecture, especially with regard to *memory uncoalescence*, *divergences* and *load imbalance*. Pangolin [14] is the only GPM system to employ GPUs. However, it has a high memory demand and has not been designed and implemented to fully utilize the GPU computing power by avoiding divergence, improving memory access pattern and mitigating load imbalance. Next we detail the main challenges concerning parallel subgraph enumeration on GPU addressed in this work.

The first challenge is the *high memory demand* imposed by enumeration. As subgraph extension relies on combining a subgraph with its extensions, it may lead to combinatorial explosion in the number of subgraphs as the enumeration progresses. A breadth-first search (BFS) style, which is used in Pangolin [14], is a natural choice for parallel subgraph enumeration as it exports a regular parallelism in the exploration of adjacency lists. However, BFS materializes all the states related to extended subgraphs, and the amount of memory required by this strategy quickly grows with the size of the subgraph, limiting its usage to enumerate only small subgraphs. On the other hand, depth-first search (DFS) style approach reduces the memory demand as only a small portion of the states (subgraphs being processed) are kept during the enumeration, but its parallel performance may be severely affected on GPU by its irregular and strided memory requests.

The second challenge arises from the inherent *irregularity* of enumeration. Current GPM systems implement parallel subgraph enumeration using a subgraph-centric processing, where subgraphs are treated as independent tasks [15]. Consequently, the thread-based parallel exploration of Figure 1 (used in Pangolin [14]), in which each thread independently explores distinct subgraphs, results in a reduced GPU performance due to the intrinsic *memory uncoalescence*, thread *divergences* and *load imbalance*. *Memory uncoalescence* and subutilization of the memory bandwidth arises as different threads within a warp access strided memory locations/graph portions according to their currently processed subgraph. For example, thread t_1 accesses positions [0-5] (dark gray positions) of graph when performing subgraph extension, while thread t_2 accesses positions [13-20] (light gray positions). *Divergence* in the execution occurs due to different sizes and processing costs of the adjacency lists (very common in scale-free graphs), leading to inefficient utilization of the GPU.

Load imbalance is the third challenge which exists as the cost of exploring subgraphs may be different and can not be known in advance. Thus, even if subgraphs s_1 and s_2 are assigned to different threads to take advantage of parallelism, idleness is likely to occur as some threads finish earlier. In this paper we propose strategies to mitigate these three challenges and allow efficient implementation of GPM algorithms on GPUs. Our contributions are summarized as:

• **DFS-wide Subgraph Exploration.** Subgraph exploration strategy designed for GPU that reduces memory demands vs. BFS, and allows massive parallel exploration of subgraphs with regular memory accesses. Our DFS-wide strategy achieved an average speedup of 12× vs DFS. To the best of our knowledge, we are the first work to use a DFS-style subgraph exploration on GPU for GPM processing;

• **Warp-centric Design.** An efficient warp-centric design of all compute demanding stages of the subgraph enumeration, improving memory coalescence and minimizing divergences;

• **Warp-Level Load-Balancing.** A load redistribution strategy among GPU warps. It is executed by CPU that monitors GPU occupancy and decides when and how load is redistributed. This layer leads to an average speedup of 16×;

• **DuMato GPU-based GPM System.** A runtime system named *DuMato*, which offers a high-level API to develop GPU GPM algorithms used to deploy our optimizations. DuMato is able to explore larger subgraphs (up to 12 vertices) and is often an order of magnitude faster than state-of-art GPM systems.

II. Background

We assume for sake of simplicity undirected graphs without labels, but our strategies may be adapted to support directed graphs and labeled features. The vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The core of our problem lies on the enumeration of *induced subgraphs*, that is, a subgraph S where, for any $v_i, v_j \in V(S)$, $(v_i, v_j) \in E(S)$ iff $(v_i, v_j) \in E(G)$.

A graph is explored through incremental visits to vertices' neighbourhood (Definition 1), called *traversals* (Definition 2). A traversal can be used to create an induced subgraph from its vertices, called *induced traversal* (Definition 2). GPM algorithms usually traverse the graph starting from each vertex/edge, and traversal strategies are usually categorized as either *breadth-first search* (*BFS*) or *depth-first search* (*DFS*). When two traversals find an *isomorphism* (Definition 3) between subgraphs sharing the same vertex set, there is an *automorphism* (Definition 3).

Definition 1: Given a graph G and a subgraph S of G, the **neighbourhood** of S is defined as $N(S) = \{v \in neighbours(u) \mid u \in V(S)\} \setminus V(S)$.

Definition 2: A **traversal** is an array $tr = [v_1, \cdots, v_k]$ of k unique vertices of a graph G ($tr \subseteq V(G)$), which stores an order each vertex $v \in tr$ is visited in G. An **induced traversal** is accompanied by the existing edges among the vertices.

Definition 3: An **isomorphism** between two graphs G and H is a bijective function $f : V(G) \rightarrow V(H)$ such that, for all edges $(v_i, v_j) \in E(G)$, $(f(v_i), f(v_j)) \in E(H)$. An isomorphism between two graphs G and H such that $V(G) = V(H)$ is an **automorphism**.

Given a graph G and the set of traversals T such that each $tr \in T$ creates the same induced traversal tr_{ind}, the *canonical candidate* is the only traversal $tr \in T$ whose visiting order of vertices is allowed to reach tr_{ind} in G. GPM algorithms generate only canonical candidates, preventing different traversals from finding the same induced traversal during exploration, avoiding redundant computation. Canonical candidates may be converted to a unique representation called *canonical representative* (also referred in this work as *patterns*). The conversion of an induced traversal to its corresponding canonical representative is known as *canonical* *relabeling*, which is an operation performed very often in GPM algorithms to categorize subgraphs.

GPM algorithms rely on enumerating k-vertex subgraphs that follow a given property. Equation 1 describes *enumeration function E*, which can be used to design GPM algorithms. Given a graph G, an initial traversal tr, and an integer k, the function E explores traversals with k vertices that satisfy a given anti-monotonic property P, producing results through an output function A. Functions P and A enable application-specific semantics. For instance, motif counting [7] would have A counting how many canonical candidates exist per canonical representative and clique counting [9] would have P selecting only canonical candidates that are fully connected.

$$E(G,tr,k) = \begin{cases} \emptyset & \text{if } |tr| = 0 \\ \bigcup\limits_{u \in N(tr)} E(G, P(tr+u), k) & \text{if } |tr| < k \\ A(tr) & \text{if } |tr| = k \end{cases} \qquad (1)$$

III. Related Work

This section presents the closest GPM systems available in the literature and Table I summarizes their main features. Graph analytics systems (e.g. cuGraph[2]) are not presented because they implement algorithms (e.g. PageRank) which tend not to experience combinatorial explosion in the number of intermediate states and allow more regular execution.

GPM System	Algorithmic approach	Proc.	Explor. strate.	Warp-centric	Load bal.
Arabesque [11]	Pattern-oblivious	CPU	BFS	n/a	✓
RStream [13]	Relational	CPU	BFS	n/a	✓
Automine [12]	Pattern-aware	CPU	DFS	n/a	×
Fractal [5]	Pattern-oblivious	CPU	DFS	n/a	✓
Peregrine [6]	Pattern-aware	CPU	DFS	n/a	×
Pangolin [14]	Pattern-oblivious	GPU	BFS	×	×
GraphZero [16]	Pattern-aware	CPU	DFS	n/a	x
DuMato	Pattern-oblivious	GPU	DFS	✓	✓

TABLE I: Related work. "n/a" stands for "not applicable".

GPM systems for CPU. *Arabesque* [11] is one of the first GPM systems targeting distributed memory machines. The algorithmic approach adopted by Arabesque is known as pattern-oblivious because it does not rely on pattern generation to guide the subgraph enumeration. Arabesque proposes a data structure to compress subgraphs in-memory and to mitigate the memory demands of the BFS-style exploration while it also employs load balancing. *RStream* [13] is a relational GPM system that relies on expensive join operations to perform subgraph enumeration. It presents limitations caused by high memory consumption as the length of enumerated subgraphs increases. *Fractal* [5] is a distributed memory CPU-based GPM system that uses a DFS exploration strategy to reduce memory demands. Fractal proposes and implements a hierarchical work-stealing mechanism to mitigate load imbalance. *AutoMine* [12] proposes an automated code generation for GPM algorithms on CPU. It employs efficient scheduling of intersect/subtract operations to automate code generation for custom patterns. Because this approach is specialized for specific patterns, it may be too expensive in general-purpose GPM scenarios, where subgraph exploration typically involves multiple patterns. *Peregrine* [6] is a parallel GPM system designed for shared-memory CPU machines. GraphZero [16] is a compilation-based GPM system which improves AutoMine's schedule generation and symmetry breaking. Both Peregrine, AutoMine and GraphZero use an exploration strategy known as pattern-aware, where canonical representatives are used to guide the subgraph enumeration by leveraging specialized execution plans. Although pattern-aware exploration is efficient for enumerating small subgraphs, it has limitations whenever the application searches for a large number of canonical representatives (e.g., counting large motifs). DuMato adopts a pattern-oblivious exploration strategy and, thus, is not limited by the number of patterns mined.

[2]https://github.com/rapidsai/cugraph

GPM systems for GPU. *Pangolin* [14] is the only GPM system designed for GPU and follows a pattern-oblivious enumeration using the BFS exploration strategy. Pangolin's design enables execution optimizations by pruning the search-space of subgraphs and by reducing the amount of isomorphism tests required. Materialized intermediate states generated by the BFS exploration facilitate the runtime to leverage BSP (*Bulk Synchronous Parallel*) load balancing schemes. However, the BFS high memory demand limits its applicability to enumerate small subgraphs. Besides, Pangolin does not leverage optimizations to handle irregularity of parallel GPM algorithms on GPU and relies on CPU frameworks to perform isomorphism tests.

As presented in Table I, DuMato is the only GPM system designed to efficiently use GPU with a DFS-like exploration strategy. Consequently, the memory demands, compared to Pangolin, are significantly smaller and larger subgraphs can be mined. Our system also computes isomorphism tests efficiently on GPU. DuMato has been optimized to use the warp-centric execution model, leading to better use of the GPU computing power due to the reduced thread warp divergence and optimized (coalesced) memory accesses. Finally, DuMato also proposes and implements a lightweight load balancing mechanism to redistribute load among GPU thread warps.

IV. Efficient Strategies for GPU Graph Pattern Mining

In this section we present our strategies for efficient graph pattern mining algorithms on GPU. We start with an overview of *DuMato*, our system that supports high-level implementation of GPM algorithms on GPU. Next, we use DuMato execution workflow to present our strategies to reduce memory demand, improve memory coalescence and divergences, and mitigate load imbalance.

A. DuMato Execution Workflow

The execution workflow of DuMato (Figure 2) employs the *filter-process* model [11], which allows implementation of GPM algorithms based on the enumeration function E (Eq. 1).

Each circle refers to a phase in the workflow and each diamond refers to a decision.

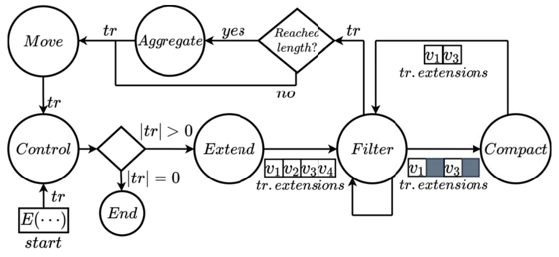


Fig. 2: DuMato execution workflow.

The process starts with a call $E(G, tr, k, P)$ to enumerate and output traversals of size k that satisfy property P extended from an initial traversal tr. The initial traversal tr is used as input to a *Control* phase, which implements the termination condition ($|tr| = 0$ in Eq. 1). The output of the *Control* phase is a decision on whether the subgraph enumeration should proceed (traversal is not empty) or terminate (traversal becomes empty). If enumeration continues, the *Extend* phase computes the extensions from the current traversal ($|tr| < k$ in Eq. 1). These possible extensions are in the neighborhood of the current subgraph and are obtained from the adjacency of vertices in the traversal (Def. 1). The *Extend* phase outputs the current traversal and its extensions.

Next, application-specific semantics may be employed to narrow the subgraph search in the *Filter* phase, which selects subgraphs that satisfy some anti-monotonic property P ($|tr| < k$ in Eq. 1). For example, a property P may check whether a subgraph is a clique. This is carried out by passing over the extensions to invalidate those that do not satisfy property P. Multiple Filters may be executed depending on which conditions must be verified to ensure property P. The Filter phase outputs the current traversal and extensions that are valid.

The output of the Filter may have several invalidated (erased) positions in the array of extensions, e.g. v_2 and v_4 in Figure 2. This array of non-contiguous valid extensions may cause substantial performance degradation, as the next Filter may have to pass over and process (or check) invalid values. Thus, DuMato proposes an optional *Compact* phase executed after each Filter to reorganize valid extensions into a contiguous memory/array.

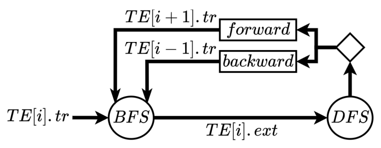
After all Filter/Compact phases were executed, if traversal size reaches the target number of vertices, they are forwarded to the enumeration output A ($|tr| = k$ in Eq. 1). This is accomplished in the *Aggregate* phase, in which traversals are consumed for counting, pattern counting, or buffering. If the traversal has not reached the target number of vertices, the Aggregate phase is skipped.

Further, the *Move* phase decides whether to move forward or backward in the subgraph enumeration. Moving forward means that an unprocessed extension is appended to the current traversal for processing (recursion call). Moving backwards means that all extensions of the current traversal have been processed, and that the algorithm can go back on processing smaller traversals (recursion return). The output of the *Move* is a modified traversal that should restart the workflow at the Control, closing the cycle in Figure 2.

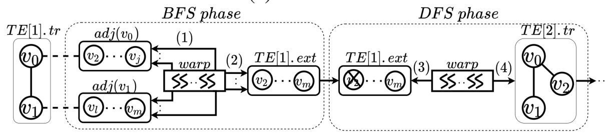
Our task allocation modeling is warp-centric and each warp receives a traversal to enumerate. Threads within a warp enumerate the same traversal cooperatively, alternating between SIMD and SISD phases throughout the execution workflow. The next sections detail the design and the implementation of each phase of the execution workflow, which shows our strategies to mitigate the memory demand (Section IV-B), the execution irregularity (Section IV-C), and the load imbalance (Section IV-D) of GPM algorithms on GPU. We also present our easily-programmable API (Section IV-E).

B. DFS-wide Subgraph Exploration

We propose a novel DFS-wide subgraph exploration, which alternates between a BFS and a DFS phase to allow regular subgraph enumeration on GPU. Figure 3a presents an overview of the DFS-wide exploration steps and Figure 3b shows the operations performed in one iteration of BFS and DFS phases. TE (*Traversal Enumeration*) is an array that stores the intermediate states needed for DFS-wide. $TE.tr$ stores the vertex ids of the current traversal and $TE.ext$ stores the extensions generated through enumeration.



(a) Overview.



(b) BFS and DFS steps.

Fig. 3: DFS-wide subgraph exploration

In Figure 3a, the enumeration starts with a traversal $TE[i].tr$ and the BFS phase produces and stores the extensions efficiently in a contiguous array ($TE[i].ext$) which will be cached. The DFS receives the extensions and decides to move forward or backward in the enumeration, depending on the length of tr and the extensions. Note that, in both forward and backward, the DFS phase will access extensions in a contiguous memory which is probably cached, improving memory efficiency. Enumeration proceeds alternating between BFS and DFS steps until the traversal reaches the target size. Assuming we want to enumerate a traversal $tr = \{v_0, v_1\}$, Figure 3b details the operations performed in BFS and DFS phases in a single iteration of DFS-wide. In the BFS phase a warp visits the adjacency lists of vertices in current traversal (step 1), copies these vertices to extensions, and keeps only the unique extensions which are not in tr (step 2). Once extensions are generated, the DFS phase starts by consuming a vertex (v_2) from extensions (step 3) and incrementing the current traversal (step 4).

The BFS phase is implemented by the warp-centric $Extend$ phase (described later in Section IV-C1) of DuMato workflow, and the DFS phase is implemented by the warp-centric *Move* phase (Algorithm 1). *Move* phase allows the warp to move forward/backward in the enumeration of a traversal. It receives TE and a flag $genedges$ to indicate whether the edges of traversals should be generated throughout enumeration. If the current traversal still has not reached the size limit and the current set of extensions is not empty (line 3), the warp moves forward in the enumeration by consuming an extension and extending the current traversal (lines 4,5). If the edges of traversal are needed, *induce* function (line 6) is a SIMD step that reuses the edges of current traversal to produce the edges of the extended traversal. If either the current traversal has reached the size limit or the current extensions set is empty, the current traversal can not be extended and the warp moves backward in the enumeration (line 7). In case the enumeration of current traversal finishes, the warp pulls a new traversal from a global queue (line 8). As all threads within a warp manipulate the same traversal and the main purpose of $Move$ is to update information about current traversal, it is mostly a SISD phase, and only *induce* function that is costly is a SIMD step.

Algorithm 1: Move primitive

```
1          $void\ move(TE,\ genedges):$
2  SISD    $extensions \leftarrow TE[TE.len-1].ext;$
3  SISD    $if(TE.len \neq k-1\ and\ extensions \neq \emptyset):$
4  SISD        $ext \leftarrow pop(extensions);$
5  SISD        $TE[level+1].tr \leftarrow ext;$
6  SIMD        $if(genedges): induce(TE);$
7  SISD    $else: TE.len--$
8  SISD    $if(TE.len = 0): TE \leftarrow pull\ trav.\ from\ global\ queue;$
```

The worst-case space complexity of the DFS-wide exploration is $O(traversals \times max(G) \times k^2)$, where $traversals$ is the number of traversals processed in parallel, $max(G)$ is the maximum degree of the input graph, and k is the length of explored subgraphs. All data structures are allocated in global memory and shared memory was set for caching, which is used in the BFS phase during the copy of adjacency lists to the extensions, as well as in the DFS phase to read an extension to move forward/backward. The cost for BFS subgraph exploration is $O(traversals \times max(G)^{k-1})$, which naturally leads to an exponential growth of memory demands as k increases. The cost for DFS subgraph exploration is $O(traversals \times k)$, as the only intermediate state needed is the set of vertex ids of current traversal. Although DFS consumes less memory than DFS-wide, DFS-wide allows more regularity in execution and memory access pattern throughout subgraph enumeration.

C. Efficient Warp-centric Filter-Process

This section describes the warp-centric based design and implementation of the DuMato phases of the filter-process workflow. Our goal with this model is to minimize execution divergence in our irregular algorithms, and to exploit the opportunities of parallelism and regular memory access enabled with the DFS-wide strategy.

1) Extend: This phase is the BFS step that generates the neighbourhood extensions of a traversal tr by visiting the adjacency lists of a specific range of vertices. This design is important to enable algorithms using the adjacency list of all vertices in the current traversal (e.g. motif counting) or only the adjacency list of a subset (e.g. clique counting).

Algorithm 2 shows our warp-centric implementation of the *extend* phase. Every call to *extend* returns a *boolean* value to indicate whether its extensions had already been filled prior to the call, and this information is useful to avoid unnecessary calls to *filter-compact*. Line 2 is an initial SISD phase, where all threads in the warp receive the array where extensions of current traversal are supposed to be written. In case the extensions have already been generated by previous calls to *extend*, the function stops and returns *false* (line 3). Lines 4-9 generate the extensions of current traversal by iterating the adjacency lists of vertices in $TE[start \cdots end]$.

Threads in the warp receive the same vertex id in the current traversal (line 4), whose adjacency will be visited in parallel. Each thread retrieves a different extension candidate e by reading the adjacency of id in parallel (line 5). As the adjacency list of a vertex is contiguous in global memory, this memory request is coalesced. Next, threads in the same warp work cooperatively to discover whether their extension candidates are valid for the current traversal. Threads compare their extension candidates to each vertex in current traversal to check whether they are already present in the current traversal (line 6). In this step, threads in the same warp execute in lockstep and compare their values to the same position i in $TE[i].tr$, allowing broadcast of $TE[i].tr$ to all threads in the warp using one memory transaction. Next, threads compare their extension candidates to extensions already generated (line 7). In this step we also take advantage of lockstep execution and memory broadcasting, providing regular execution and reducing memory transactions. In case the extension candidates neither are in the current traversal nor in the extensions already generated, they are written to the current extensions in parallel through coalesced memory writes (lines 8-9). Note that all lines of *extend* function are executed in lockstep by threads within a warp, minimizing divergences. Besides, each line also provides regular memory access patterns for all data structures, allowing memory coalescence and good cache locality. If new extensions were generated, the function returns $true$ (line 10).

2) Filter: Given a traversal tr, *filter* (Algorithm 3) phase iterates a set of extensions in parallel and invalidates those that do not meet a property P. Threads read consecutive extensions in parallel through coalesced memory accesses (line 3), call a function P to discover whether an extension satisfies the desired property (line 4), and invalidate (write -1 value)

Algorithm 2: Extend primitive.

```
1         $boolean\ extend(TE,\ start,\ end):$
2  SIMD     $len \leftarrow TE.len - 1$ ; $extensions \leftarrow TE[len].ext;$
3  SISD     $if(extensions\ generated)$: **return** $false;$
4  SISD     $for\ each(id \in TE[start \cdots end - 1].tr):$
5  SIMD       $for\ each(e \in adjacency(id)):$
6  SIMD         $inTraversal \leftarrow find\ e\ in\ TE[0 \cdots len].tr;$
7  SIMD         $inExtensions \leftarrow find\ e\ in\ extensions;$
8  SIMD         $e \leftarrow !inTraversal\ \&\&\ !inExtensions\ ?\ e\ :\ -1;$
9  SIMD         $write\ e\ to\ extensions;$
10 SISD     **return** $true;$
```

those which do not fulfill the property (line 4). We provide warp-aware functions for implementing P and accessing TE data structure, including filtering and searching routines over extensions/adjacencies. Even if a user implements P without our utility library, our filtering primitive implementation calls *__syncwarp()* after P (line 4 in Alg. 3) to guarantee warp-centric execution for the rest of the workflow.

Algorithm 3: Filter primitive.

```
1         $void\ filter(TE,\ P,\ args):$
2  SISD     $extensions \leftarrow TE[TE.len - 1].ext;$
3  SIMD     $for\ each(ext \in extensions):$
4  SIMD       $if(P(TE, ext, args)): invalidate(extensions, i);$
```

3) Compact: It is an optional phase that accesses the extensions set of the current traversal and removes invalid positions, reducing its actual length. As seen in Algorithm 3, a filter iterates over the entire set of extensions of the current traversal, even if some positions are invalid. By removing invalid positions, compaction reduces the costs of sequential filter calls. We provide an efficient warp-centric implementation of this function using intra-warp communication primitives such as *ballot* and *any_sync*.

4) Aggregate: This phase is executed when a thread warp has derived traversals with k vertices and, as discussed, it is in charge of producing the actual GPM algorithm results (A function of Eq. 1). *DuMato* provides three aggregation primitives: *aggregate_pattern*, *aggregate_counter* and *aggregate_store*, as defined in Table II, which are explained below.

The *aggregate_pattern* is the most challenging primitive for implementation on GPUs. It is used when the output of the GPM algorithm relies on counting the occurrence of canonical representatives (patterns) with k vertices, such as motif counting. This is executed in a warp basis such that each warp performs *canonical relabeling*, converting each subgraph with k vertices to its canonical representative and incrementing a counter. This is only possible due to our novel representation for canonical representatives, which reduces the amount of memory required to store them. The solution for canonical relabeling relies on graph isomorphism, and GPM systems (including Pangolin [14]) perform it on CPU using tools such as Nauty [17]. To the best of our knowledge, we are the first work to implement canonical relabeling on GPU.

Figure 4 depicts our strategy for canonical relabeling on GPU. We use a bitmap to store the edges of the traversal. For example, assuming $k = 4$ and a traversal tr, we need 5 bits to store the edges of a traversal. As we handle only connected traversals, v_0 is always connected to v_1 and this edge is not stored. The two least significant bits of the bitmap store the edges of v_2 with respect to $\{v_0, v_1\}$, and the next three bits store the edges of v_3 with respect to $\{v_0, v_1, v_2\}$ (the same reasoning may be applied to a subgraph with k vertices). Using 5 bits we can represent up to 32 possible traversals, as seen in (a). Each possible traversal with 4 vertices can be mapped to its canonical representative, shown in (b). As traversals often produce isomorphic subgraphs, different traversals may be mapped to the same canonical representative. The amount of canonical representatives is much smaller than the amount of possible traversals, as seen in (c), and the bitmap representation of canonical representatives can be relabeled to use consecutive bitmaps.

Our implementation creates a dictionary that receives a traversal *tr* with k vertices along with its edges encoded using the bitmap representation (a.k.a. an induced traversal) and converts *tr* to a canonical representative that is in a contiguous range of positions (Figure 4). This is done in two steps: in $(a) \rightarrow (b)$ traversal edges are mapped to non-contiguous representatives; and in $(b) \rightarrow (c)$ non-contiguous representatives are mapped to contiguous identifiers. This conversion allows each warp to use local counters for canonical representatives using less memory, as no position in the array of counters is wasted. This dictionary is a pre-processed data structure, created once for a range of k values, and that can be used in any dataset and in any application that requires canonical relabeling (e.g. frequent subgraph mining [18] and subgraph matching [10]). DuMato provides this dictionary as an input file.

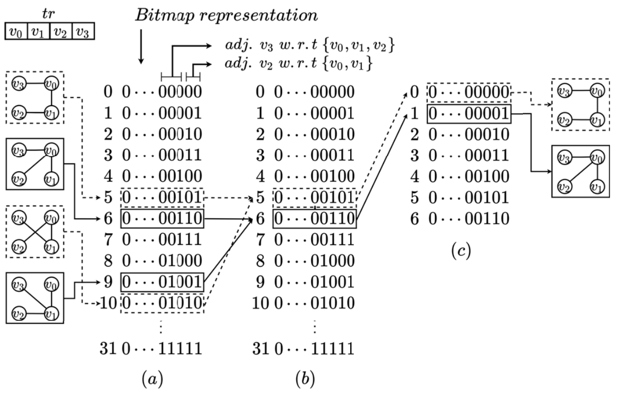


Fig. 4: Canonical relabeling on GPU.

The *aggregate_counter* primitive is called when the desired results/output of the GPM algorithm is a pattern counting, such as in the clique counting algorithm. Each warp produces its own counter (based on the length of the extensions for each traversal with $k - 1$ vertices) to avoid inter-warp race conditions, and the global counting is produced with a reduction of

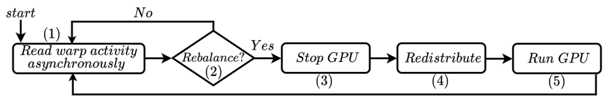


Fig. 5: Warp-level load balancing.

the warps counting afterwards, on CPU. This is a simple and computing inexpensive primitive. Primitive $aggregate\_store$ stores the explored subgraphs with k vertices and can be used in algorithms such as subgraph querying, which lists all subgraphs that matches a pattern instead of producing counters. We create an array buffer that stores the connectivity bitmap of explored subgraphs with k vertices as they are produced. DuMato then provides a producer-consumer environment using the CPU to consume the buffer asynchronously.

D. Warp-Level Load Balancing

The cost of enumerating distinct traversals may vary, which leads to load imbalance among warps. We propose an asynchronous workload redistribution scheme running on CPU (Figure 5) to mitigate this problem, which makes decisions based on the warp level activity information.

The CPU constantly and asynchronously reads the warp activity information from the GPU to decide whether load should be redistributed to improve GPU utilization (step 1). When CPU detects load balancing is to be performed (step 2, *rebalance*), the CPU informs the GPU by setting a flag and the warps stop their execution in a consistent state (step 3). We propose a *rebalance* condition such that, if the number of active warps is found to be lower than a threshold, the workload balancing is carried out. When all warps stop, CPU copies TE data structure, performs work redistribution (step 4, *redistribute*) and runs kernel again on GPU (step 5). We propose a *redistribute* algorithm that performs load balancing by separating warps in *donators* (those with multiple traversals) and *idle* ones. While there are *idle* warps, a donator is selected (in a round-robin fashion) and an arbitrary traversal is migrated from it to the idle warp. As long as GPU is active, CPU inspects warp activity and perform work redistribution. *Rebalance* and *redistribute* steps in our scheme can be easily customized to implement other approaches. Our CPU-only strategy mitigates synchronization overheads from GPU and more resources can be allocated for subgraph enumeration.

E. Programming with DuMato

DuMato's workflow (Fig. 2) is able to represent any GPM algorithm relying on the enumeration of induced subgraphs. Table II shows the programming interface that can be used to create algorithms in DuMato using our efficient strategies. The functions receive as parameter a data structure holding runtime information about the active traversal and extension arrays (TE, detailed in Fig. 3) along with additional parameters.

Control and *Move* phases are responsible for keeping the workflow active while there are unprocessed traversals in the search space. Because this loop-based exploration is common

Functions	Phase	Scope
`[CT]` *control(TE)*	Control	Algorithm-independent
`[MV]` *move(TE, genedges)*	Move	
`[EX]` *extend(TE, begin, size)*	Extend	Algorithm-specific
`[FL]` *filter(TE, P, args)*	Filter	
`[CP]` *compact(TE)*	Compact	
`[A1]` *aggregate_counter(TE)*	Aggregate	
`[A2]` *aggregate_pattern(TE)*		
`[A3]` *aggregate_store(TE)*		

TABLE II: DuMato API.

to most GPM algorithms searching for multiple subgraphs, we say that these phases are independent of algorithm semantics. Functions `[CT]` and `[MV]` implement these two phases. `[CT]` allows the underlying runtime to check the termination conditions of the execution. `[MV]` implements the traversal order of exploration and receives an additional parameter $genedges$ that determines whether the edges of the current traversal should be generated.

Extend, *Filter*, *Compact*, and *Aggregate* phases enables a straightforward and efficient representation of application-specific semantics on GPUs. Function `[EX]` implements the *Extend* phase and generates the extensions array by fetching the neighborhood of vertices in the traversal at positions in range $[begin, size)$. This can be used to generate extensions using alternative strategies that may be more effective to explore subgraphs having patterns known apriori [6], [9]. `[FL]` implements the *Filter* phase and allows invalidating extensions that do not satisfy a user-defined property. The input to this call is a function and its arguments (property P and $args$, respectively) that is applied to each extension to maintain only valid ones. This interface can be used to design custom subgraph filters of extensions based on canonical candidate generation [11], density [19], subgraph matching [10], among others. `[CP]` implements the *Compact* phase and can be applied between consecutive `[FL]` calls to compact the extensions array. This interface allows a fine-grained control over the underlying GPU memory organization, which can be useful to speedup memory access when it is possible to infer the selective potential of filters from application-specific semantics [9]. `[A1]`, `[A2]`, and `[A3]` implement the *Aggregate* phase: `[A1]` counts the number of valid extensions in the array of extensions; `[A2]` counts the number of traversals per pattern; and `[A3]` allows buffering of traversals for custom semantics and further downstream processing. These can be used, for instance, for subgraph counting [9] and scoring [20].

Algorithms are implemented in a loop that processes new traversals until the termination condition is reached. After each loop iteration, DuMato moves the exploration to a new traversal as a preparation to the next iteration. This is common to most GPM algorithms and can be observed in lines 10 and 17 of Algorithm 4, which presents the implementation of two representative GPM algorithms using DuMato API: *clique counting* and *motif counting*. Bold lines marked with 👤 represent algorithm-specific semantics that uses DuMato's

API and, consequently, new algorithms with new extend, filtering, and aggregation demands may be implemented by replacing those lines.

Algorithm 4: Clique and motif counting algorithms.

```
1      $void\ clique\_counting(TE):$
2        $while(control(TE)):$
3  👤      $\mathbf{if(extend(TE, 0, 1)):}$
4  👤        $\mathbf{u \leftarrow TE[TE.len - 1].id;}$
5  👤        $\mathbf{filter(TE, \&lower , [u]);}$
6  👤        $\mathbf{compact(TE);}$
7  👤        $\mathbf{filter(TE, \&is\_clique, []);}$
8          $if(TE.len = k - 1):$
9  👤        $\mathbf{aggregate\_counter(TE);}$
10         $move(TE, false);$

11     $void\ motif\_counting(TE):$
12       $while(control(TE)):$
13 👤      $\mathbf{if(extend(TE, 0, TE.len)):}$
14 👤        $\mathbf{filter(TE, \&canonical, []);}$
15         $if(TE.len = k - 1):$
16 👤        $\mathbf{aggregate\_pattern(TE);}$
17         $move(TE, true);$
```

Clique counting. Given a graph G, the clique counting problem seeks to count the number of cliques with k vertices within G. Clique counting represents algorithms whose goal is searching for subgraphs with the same pattern. Because a clique extension must be adjacent with every vertex in the traversal, the *Extend* phase consists of generating the array of extensions from the neighbors of a single vertex in the traversal. `[EX]` call in line 3 of Algorithm 4 implements this idea by indicating that the current extensions should be obtained from the neighbors of the first vertex in the traversal (represented by the range $[0, 1)$). Given this set of extensions, `[FL]` is used in line 5 to invalidate non-canonical candidates (extensions lower than the last vertex), `[CP]` is used in line 6 to reorganize the extensions array by compaction, and `[FL]` is used again in line 7 to remove extensions that do not generate cliques. The custom procedure is_clique ensures that valid extensions are connected to all vertices in the traversal. Both $lower$ and is_clique are simple functions that must return $true$ or $false$ given a traversal and one of its extensions. Finally, if the traversal reaches $k-1$ vertices then traversals with k vertices may be aggregated with `[A1]`, which accumulates the length of the array of extensions in a counter.

Motif counting. A motif of size k is a canonical representative subgraph containing k vertices. The motif counting problem seeks to count the number of each motif of size k in a graph G. Motif counting represents algorithms whose target is searching for subgraphs of multiple patterns. Because this problem requires visiting all induced subgraphs of size k, the `[EX]` call in line 13 indicates that the adjacency of each vertex in the traversal must be considered to produce the extensions array (i.e. all traversal vertices in range $[0, TE.len)$). In line 14 the algorithm calls `[FL]` to invalidate extensions that combined with the traversal do not represent canonical candidates. Custom function $is\_canonical$ can be implemented using standard *canonical filtering algorithms* [11]. Finally, a `[A2]` call extracts the canonical representative (pattern) from traversals combined with last level extensions to increment the respective pattern-specific counters (line 16).

V. Evaluation

This section presents the DuMato performance evaluation. We employ the implementations of *clique counting* and *motif counting* algorithms. The algorithms represent two important categories in GPM processing: exploration of subgraphs sharing a single canonical representative (Clique counting) and exploration of subgraphs ranging multiple canonical representatives (Motif counting). The attributes of five real-world datasets used in our experiments are presented in Table III. CPU experiments were conducted on an Amazon AWS machine with 16 vCPUs optimized for CPU computing (*C5a.4xlarge*), 32GB of RAM and Ubuntu 22.04. GPU experiments concerning execution time were conducted on an Amazon AWS machine with one NVIDIA Tesla V100 (*p3.2xlarge*) with 32 Gb and CUDA 11. GPU profiling experiments were conducted on a local machine with NVIDIA TITAN V with 12GB 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%). After a theoretical occupancy analysis and an empirical evaluation, the configuration of the experiments was set to 172,032 threads for all datasets. For the motif counting we do not present the results for LiveJournal graph because the executions exceed our 24 hours limit even for small subgraph sizes ($k > 4$).

Dataset	V(G)	E(G)	Avg. Deg.	Density	Max. Deg.
Citeseer [18]	3.2K	4.5K	2.77	8.51×10^{-4}	99
ca-AstroPh [21]	18.7K	198.1K	21.10	1.12×10^{-3}	504
Mico [18]	96.6K	1.08M	22.35	2.31×10^{-4}	1359
com-DBLP [22]	317K	1.04M	6.62	2.08×10^{-5}	343
com-LiveJournal [22]	3.9M	34.6M	17.35	4.34×10^{-6}	14815

TABLE III: Graphs used for evaluation.

A. *Impact of Optimizations*

This section evaluates the efficiency of our optimization strategies using three implementations of clique and motif counting (all implemented with DuMato API): *DM_DFS* (***DuM**ato **D**epth-**F**irst **S**earch*), in which each GPU thread receives a traversal tr and calculates $E(G, tr, k, P)$ using standard DFS exploration; *DM_WC* (***DuM**ato **W**arp-**C**entric*), in which each warp receives a traversal and enumerates it using our novel DFS-wide approach and the warp-centric design, but with load balancing disabled; *DM_OPT* (***DuM**ato **Opt**imized*), which is *DM_WC* with load balancing enabled. Table IV shows the execution times for the three versions of both algorithms as the length of the subgraphs mined (k) is varied. Cells containing "-" refer to experiments that have not finished within 24 hours.

1) Gains Due to Warp-centric DFS-wide: The DFS version consumes a small amount of memory, but each thread within a warp has its own execution path, leading to an irregular execution and a worse memory access pattern. The *DM_WC* version increases both memory efficiency and parallelism regularity, achieving speedups up to 33x (Clique app, Mico dataset and $k = 5$) compared to *DM_DFS*, and showing the efficiency of memory coalescence and divergence reduction.

To understand the effects of our exploration and optimization strategies at hardware level, Table V shows the improve-

		Impl.	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
Clique	Citeseer	DM_DFS	0.01	0.01	0.01	0.01	∅
		DM_WC	0.01	0.01	0.02	0.02	∅
		DM_OPT	0.01	0.01	0.02	0.03	∅
	ca-AstroPh	DM_DFS	0.23	4.75	51.43	430.11	44.78K
		DM_WC	0.03	0.36	3.50	28.98	221.75
		DM_OPT	0.13	0.28	0.67	2.37	11.46
	Mico	DM_DFS	3.28	267.32	19.67K	-	-
		DM_WC	0.26	12.62	593.94	26.31K	-
		DM_OPT	0.33	1.93	51.98	1.75K	-
	DBLP	DM_DFS	0.16	4.04	134.13	3.64K	-
		DM_WC	0.03	0.33	8.04	232.96	5.63K
		DM_OPT	0.13	0.28	1.01	7.14	96.22
	LiveJournal	DM_DFS	337.85	6.65K	-	-	-
		DM_WC	16.83	260.25	6.77K	-	-
		DM_OPT	4.30	49.82	897.25	38.50K	-
Motifs	Citeseer	DM_DFS	0.01	0.49	10.84	232.11	6.11K
		DM_WC	0.01	0.06	1.26	25.90	457.17
		DM_OPT	0.11	0.11	0.23	0.68	8.27
	ca-AstroPh	DM_DFS	1.59	555.64	-	-	-
		DM_WC	0.09	20.26	5.28K	-	-
		DM_OPT	0.13	1.78	149.43	28.14K	-
	Mico	DM_DFS	20.58	13.90K	-	-	-
		DM_WC	0.95	597.66	-	-	-
		DM_OPT	0.46	33.44	10.56K	-	-
	DBLP	DM_DFS	0.96	178.69	26.69K	-	-
		DM_WC	0.08	8.98	1.35K	-	-
		DM_OPT	0.14	1.05	38.07	2.95K	-

∅: no valid subgraphs

TABLE IV: Optimizations performance. Execution time (seconds) for three implementations of algorithms using DuMato.

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

TABLE V: Improvements of DM_WC over DM_DFS.

ments of *DM_WC* over *DM_DFS* using execution and memory metrics collected from CUDA NVProf profiling tool [23]. We present the results using DBLP dataset for k up to 4 (GPU profiling is much slower than standard runs). Metrics are divided into two categories: (i) Execution, which measures the efficient use of GPU execution model and parallelism and (ii) Memory, which quantifies the use of the memory hierarchy. For execution, we chose the metric *inst_per_warp*, which calculates the average number of instructions executed by each warp. The more regular the execution is, the less divergent instructions are issued and warps require less instructions. For memory, we chose the metric *gld_transactions*, which measures the total amount of load transactions requested to global memory. The more coalesced is the memory access pattern, the less transactions are needed to service memory requests. In our experiments, we observed that the other metrics were consistent with these two representative choices.

Execution metrics: The Warp-Centric DFS-Wide exploration results in natural lockstep implementation, which fits better GPU execution model and allows all threads within a warp to execute the same instruction more often to minimize divergence. This reduces the total number of instructions per warp for the *DM_WC* version, as they execute mostly in lockstep and all threads in the warp tend to execute the same instruction. This regularity is confirmed by the execution metrics, with improvements ranging from 3.8x and 13.3x, confirming that our warp-centric design provides more regular execution.

Memory metrics: The Warp-Centric DFS-Wide exploration with its regular lockstep execution allowed threads to perform memory requests together using coalesced requests. Therefore, our *DM_WC* version reduces the total amount of memory transactions. This reduction is confirmed by the memory metric, with improvements ranging from 2.90x to 7.92x, enhancing that our memory optimizations reduce wasted bandwidth and improve memory efficiency.

2) Improvements with Load Balancing: In order to define the adequate load balancing threshold, we conducted a sensitivity analysis (not shown due to space constraints) varying the amount of threads and the threshold used for rebalancing. We found that $172,032$ threads were enough to provide massive parallelism without overloading register allocation. We also found that, for this amount of threads, the optimum load balancing threshold was 40% for clique counting and 10% for motif counting. As clique counting prunes the search space, load imbalance occurs earlier than in motif counting, requiring a larger value of threshold to improve efficiency of the load balancing layer.

Table IV shows that the *DM_OPT* version attained speedups of up to 65× compared to *DM_WC* (Motifs app, *Citeseer* dataset and $k = 8$). As the size of enumerated subgraphs increases, work skewness is intensified because most subgraphs are extracted from denser regions of the graph associated with increasingly fewer vertices and, at this point, load balancing becomes more effective. Hence, *DM_OPT* allowed the exploration of larger subgraphs for all datasets. Whenever the amount of work is insufficient to exhibit a substantial imbalance or to payoff the overhead of redistributing the load ($k \leq 4$ in small datasets), *DM_WC* outperforms *DM_OPT*.

B. Comparison to Other GPM Systems

This section compares our optimal DuMato GPU implementations (*DM_OPT*) against three representative state-of-the-art GPM systems: Pangolin [14] GPM system designed for GPU, and Fractal [5] and Peregrine [6] parallel CPU machines. Table VI also shows the results. DuMato is more scalable and able to explore larger subgraphs than all baselines within the same time limit, exploring subgraphs of up to 12 vertices. To the best of our knowledge, this length of explored subgraphs has not been accomplished by any other GPM system searching for exact outputs.

Pangolin clearly suffers from scalability issues. Although it achieves good performance for small datasets and small enumerated subgraphs, it usually runs out of memory when the length of explored subgraphs is close to 5 vertices, limiting its applicability and the discoveries of GPM algorithms. As compared to Fractal, we obtain significant speedups in all executions with gains ranging from 17× to 103×. In general, as length of explored subgraphs increases, the processing cost is higher and DuMato can take more advantage of GPU's massive parallel processing to achieve better gains.

Regarding Peregrine, DuMato is competitive for small values of explored subgraphs (up to 5 vertices), and shows

		System	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$	$k=12$
Clique	Citeseer	**DM**	0.01	0.01	0.01	0.01	∅	∅	∅	∅	∅	∅
		FRA	4.84	4.83	4.75	4.81	∅	∅	∅	∅	∅	∅
		PER	0.01	0.03	0.02	0.02	∅	∅	∅	∅	∅	∅
		PAN	0.01	0.01	0.01	0.01	∅	∅	∅	∅	∅	∅
	ca-AstroPh	**DM**	0.13	0.28	0.67	2.37	11.46	57.55	297.89	1.47K	6.73K	28.14K
		FRA	8.17	9.75	15.89	78.09	439.16	2.30K	12.89K	57.02K	-	-
		PER	0.01	0.10	0.83	6.38	43.56	272.42	1.55K	7.93K	36.26K	-
		PAN	0.01	0.01	0.02	0.11	0.61	OOM	OOM	OOM	OOM	-
	Mico	**DM**	0.42	3.89	57.49	2.22K	-	-	-	-	-	-
		FRA	14.17	48.53	1.44K	56.72K	-	-	-	-	-	-
		PER	0.09	1.81	82.67	3.66K	-	-	-	-	-	-
		PAN	0.01	0.05	2.93	OOM	-	-	-	-	-	-
	DBLP	**DM**	0.13	0.28	1.01	7.14	96.22	1.45K	20.86K	-	-	-
		FRA	13.44	14.32	22.72	186.97	2.52K	35.51K	-	-	-	-
		PER	0.11	0.16	1.36	25.92	531.88	9.35K	-	-	-	-
		PAN	0.01	0.01	0.03	0.50	OOM	OOM	OOM	OOM	OOM	OOM
	LiveJournal	**DM**	4.30	49.82	897.25	38.50K	-	-	-	-	-	-
		FRA	394.85	901.05	16.06K	-	-	-	-	-	-	-
		PER	3.91	26.66	1.06K	64.74K	-	-	-	-	-	-
		PAN	0.01	0.53	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
Motifs	Citeseer	**DM**	0.11	0.11	0.23	0.68	8.27	157.97	-	-	-	-
		FRA	5.17	5.20	5.69	12.44	163.48	-	-	-	-	-
		PER	0.01	0.01	0.05	3.47	537.66	-	-	-	-	-
		PAN	0.01	0.01	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	ca-AstroPh	**DM**	0.13	1.78	149.43	28.14K	-	-	-	-	-	-
		FRA	9.13	435.64	4.72K	-	-	-	-	-	-	-
		PER	0.01	0.57	132.90	52.80K	-	-	-	-	-	-
		PAN	0.01	0.21	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	Mico	**DM**	0.46	33.44	10.56K	-	-	-	-	-	-	-
		FRA	16.43	474.46	-	-	-	-	-	-	-	-
		PER	0.06	6.57	7.92K	-	-	-	-	-	-	-
		PAN	0.01	3.31	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	DBLP	**DM**	0.14	1.05	38.07	2.95K	-	-	-	-	-	-
		FRA	14.33	37.62	1.43K	-	-	-	-	-	-	-
		PER	0.07	0.95	78.59	50.95K	-	-	-	-	-	-
		PAN	0.01	0.17	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM

DM: DuMato (this work); FRA: Fractal; PER: Peregrine; PAN: Pangolin
OOM: out-of-memory; *INC*: finished with incomplete results; ∅: no valid subgraphs

TABLE VI: Comparative performance. Execution time (seconds) of DuMato and baselines (GPU and CPU).

speedups of up to 65x for larger explored subgraphs. Even in Peregrine's best case (clique application, which contains only one pattern), DuMato is able to deliver consistent speedups. We achieve more expressive gains in motif counting application for larger values of k, which can be explained by the inherent characteristics of pattern-aware enumeration of Peregrine. As we increase the length of explored subgraphs, the number of valid patterns and exploration plans grows exponentially, incurring in two aspects that impact the Peregrine performance: (i) the cost of generating exploration plans for each pattern increases and (ii) part of exploration plans does not generate valid subgraphs, leading to wasted computational resources.

VI. Conclusion

We propose the DuMato GPU based GPM system that integrates novel strategies to address the challenges found for HPC of GPM algorithms on GPUs: *high memory demand*, *irregularity* (*memory uncoalescence and divergences*), *and load imbalance*. These strategies and optimizations include our DFS-wide subgraph exploration (reducing memory demand), warp-centric system design and implementation (improving memory coalescence and reducing divergences), and warp-level load balancing strategy. Our system and optimizations were evaluated using real-world datasets. We compared DuMato to three state-of-the-art GPM systems, showing that it is more scalable (up to 12-vertice subgraphs) and often one order of magnitude faster.

As a future work, we plan to extend our load balancing to allow work redistribution without having to stop and restart the GPU kernel. We also intend to propose a multi-GPU version of DuMato to accelerate it further.

Acknowledgment

This work was partially funded by Fapemig, CNPq, CAPES, and by projects CNPq/AWS (440.088/2020-8), INCTCyber (MCT/CNPq 465714/2014-5) and MASWeb (FAPEMIGPRONEX APQ-01400-14). The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer.

References

[1] Alexandra Duma and Alexandru Topirceanu. A network motif based approach for classifying online social networks. SACI '14, 2014.
[2] Olaf Sporns and Rolf Kötter. Motifs in brain networks. *PLOS Biology*, 2004.
[3] W. Lin, X. Xiao, X. Xie, and X. Li. Network motif discovery: A gpu approach. ICDE '15, 2015.
[4] Parallel discovery of network motifs. *Journal of Parallel and Distributed Computing*, 2012.
[5] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. SIGMOD '19, 2019.
[6] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. EuroSys '20, 2020.
[7] Pedro Ribeiro and Fernando Silva. G-tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov.*, 2014.
[8] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. SC '16, 2016.
[9] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. WWW '18, 2018.
[10] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. SIGMOD, 2020.
[11] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. SOSP '15, 2015.
[12] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. SOSP '19, 2019.
[13] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. OSDI'18, 2018.
[14] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 2020.
[15] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. EuroPar '14, 2014.
[16] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 2021.
[17] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 2014.
[18] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *VLDB Endow.*, 2014.
[19] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. ECMLPKDD '08, 2008.
[20] Bryan Hooi, Kijung Shin, Hemank Lamba, and Christos Faloutsos. Telltail: Fast scoring and detection of dense subgraphs. AAAI '20, 2020.
[21] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 2007.
[22] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. MDS '12, 2012.
[23] NVIDIA Corporation. Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2022.