UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Samuel Benjoino Ferraz Aquino

Strategies for Efficient Subgraph Enumeration on GPUs

Belo Horizonte
2023

Samuel Benjoino Ferraz Aquino

Strategies for Efficient Subgraph Enumeration on GPUs

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Prof. Dr. Wagner Meira Júnior
Co-Advisor: Prof. Dr. George Luiz Medeiros Teodoro

Belo Horizonte
2023

Aquino, Samuel Benjoino Ferraz.

A657s Strategies for efficient subgraph enumeration on GPUs [recurso eletrônico] : / Samuel Benjoino Ferraz Aquino – 2023.
1 recurso online (95 f. il, color.) : pdf.

Orientador: Wagner Meira Júnior.
Coorientador: George Luiz Medeiros Teodoro.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f.90-95

1. Computação – Teses. 2. Mineração de dados (Computação) – Teses. 3. Algoritmos de computador – Teses. I. Meira Júnior, Wagner. II. Teodoro, George Luiz Medeiros. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. IV. Título.

CDU 519.6*72(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Strategies for Efficient Subgraph Enumeration on GPUs.

SAMUEL BENJOINO FERRAZ AQUINO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Wagner Meira Júnior - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. George Luiz Medeiros Teodoro - Coorientador
Departamento de Ciência da Computação - UFMG

PHILIPPE OLIVIER ALEXANDRE NAVAUX:05548012053
Assinado de forma digital por PHILIPPE OLIVIER ALEXANDRE NAVAUX:05548012053
Dados: 2023.11.21 19:06:49 -03'00'

Prof. Philippe Olivier Alexandre Navaux
Instituto de Informática - UFRGS

[illegible]

Profa. Alba Cristina Magalhães Alves de Melo
Departamento de Ciência da Computação - UnB

[illegible]

Prof. Fernando Magno Quintão Pereira
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Renato Antônio Celso Ferreira
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 26 de outubro de 2023.

I dedicate this work to God, my family and friends. Specially to my father, in my loving memory.

Acknowledgments

I devote the end of this journey primarily to God, who gave me strength and resilience to overcome the challenges even when I believed I couldn't. I want to thank my parents, whose teachings allowed me to pursue my goals with faith and patience. Special thanks to my father, who would be very proud of this accomplishment. I want to thank my wife, Vanessa, who always believed in me and took care of our family and kids when I was away. I want to thank my kids Gustavo and Lara for being my emotional fuel and motivation to finish this journey. Many thanks to my dear friends for advice and fun conversations: Júlio, Vinícius Dias, Vanilson, Marcelo, Brivaldo and Carlos Teixeira. I want to thank you my advisor Prof. Wagner Meira Jr. and my co-advisor Prof. George Teodoro for all the support and patience through these years. You taught me the foundations of relevant research, which I will carry throughout my career. Finally, I would like to acknowledge the funding agencies CNPq, CAPES and FAPEMIG for financial support and the computational resources of the SDumont supercomputer provided by the National Laboratory for Scientific Computing (LNCC/MCTI).

Resumo

Mineração de padrões em grafos (MPG) é uma área em constante evolução e que compreende algoritmos com alto custo computacional. Algoritmos de MPG são construídos a partir da enumeração de subgrafos, que visita um grafo de entrada e retorna os subgrafos que atendem uma propriedade desejada. *Graphics Processing Units* (GPUs) tem sido amplamente utilizadas para acelerar algoritmos em diversas áreas. Entretanto, a enumeração de subgrafos apresenta um padrão irregular de processamento, e sua implementação em GPUs é ineficiente por conta de acessos não coalescidos de memória, divergências e desbalanceamento de carga. As estratégias existentes na literatura para a paralelização da enumeração de subgrafos em GPU são limitadas e não representam uma solução completa para todos os desafios desse problema nesta arquitetura. Esta tese propõe estratégias para projetar e implementar a enumeração de subgrafos de maneira eficiente em GPUs. Nossa estratégia de exploração *DFS-wide* reduz o consumo de memória e oportuniza otimizações no padrão de acesso a memória, que aliada ao fluxo de execução warp-centric, minimizam as divergências e melhoram o uso da capacidade computacional da GPU. Também propomos uma camada de balanceamento de carga de baixo custo para melhorar a utilização da GPU. Nossas estratégias foram implementadas em um sistema de enumeração chamado *DuMato*, que provê uma API para implementar eficientemente algoritmos de MPG. Nossa avaliação experimental mostrou que os algoritmos de MPG implementados com DuMato são até duas ordens de magnitude mais rápidos que os algoritmos implementados utilizando sistemas de MPG do estado da arte, além de serem capazes de minerar subgrafos maiores.

Palavras-chave: GPU. Enumeração de Subgrafos. Processamento Irregular. Balanceamento de Carga.

Abstract

Graph Pattern Mining (GPM) is an important and rapidly evolving area which demands high computation-demanding algorithms. GPM algorithms rely on subgraph enumeration, extracting subgraphs from an input graph that matches a given property. Graphics Processing Units (GPUs) have been an excellent platform for accelerating algorithms in many areas. However, the irregularity of subgraph enumeration makes it challenging for efficient execution on GPUs due to typical uncoalesced memory access, divergence, and load imbalance. These aspects have not been extensively addressed in previous work on subgraph enumeration using GPU. This thesis proposes strategies to design and implement subgraph enumeration efficiently on GPU. We propose a depth-first search style search (DFS-wide) that reduces memory demand while enabling sufficient parallelism to utilize the GPU, along with a warp-centric design that minimizes execution divergence and improves utilization of the GPU computing capabilities. We also propose a low-cost load-balancing layer to mitigate the inherent load imbalance of parallel subgraph enumeration. Our strategies have been implemented in a system named DuMato, which provides a simple programming interface to implement GPM algorithms efficiently. We perform an extensive evaluation of all proposed strategies, comparing DuMato with the state-of-the-art subgraph enumeration systems. Our evaluation has shown that DuMato is up to two orders of magnitude faster and can mine larger subgraphs.

Keywords: GPU. Subgraph Enumeration. Irregular Processing. Load Balancing.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

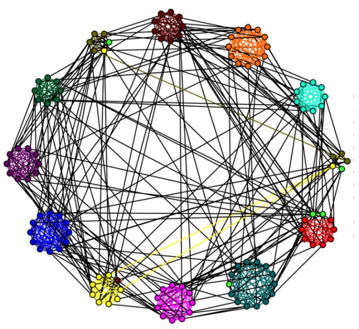
Graph pattern mining (GPM) algorithms (e.g., clique listing, motif counting, frequent subgraph mining, and subgraph matching) aim to unveil relevant subgraph patterns in graphs [60]. They are used in different domains and applications [17, 53], and Figure 1.1 depicts three examples. Figure 1.1(a) shows a graph with several communities detected using the clique listing algorithm. Figure 1.1(b) illustrates a computer vision application that uses the subgraph matching algorithm to detect image patterns. Figure 1.1(c) depicts the discovery of relevant scientific collaboration patterns using the frequent subgraph mining algorithm.

These algorithms rely on subgraph enumeration over an input graph, depicted in Figure 1.2. Subgraph enumeration visits all subgraphs of an input graph G that fullfil a graph property P required by a GPM algorithm. This property may be topological (e.g., clique) or statistical (e.g., pattern frequency [8]). The GPM algorithm produces its output (usually counting, listing or specific aggregations) as the desired subgraphs are visited by subgraph enumeration.

Suppose a GPM algorithm that needs to visit an input graph G to list all paths with three vertices. Figure 1.3 depicts how this GPM algorithm relies on subgraph enumeration. A traversal represents an order that vertices of G are visited, and the i^{th} level contains several traversals with i vertices. For example, the second level of Figure 1.3 contains 18 traversals ($\{1,5\}$,$\{1,6\}$,$\cdots$,$\{8,4\}$). Subgraph enumeration involves a combinatorial

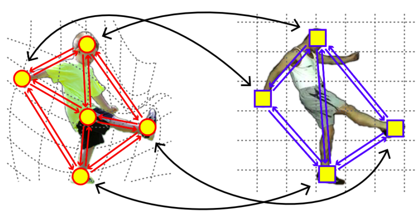
Figure 1.1: GPM applications.

(a) Community detection.



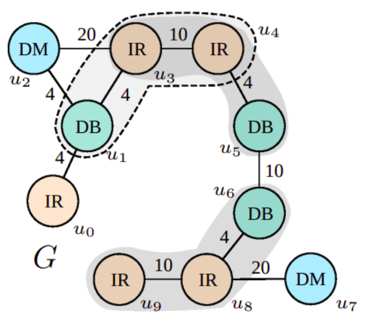
Source: [38].

(b) Computer vision [62].



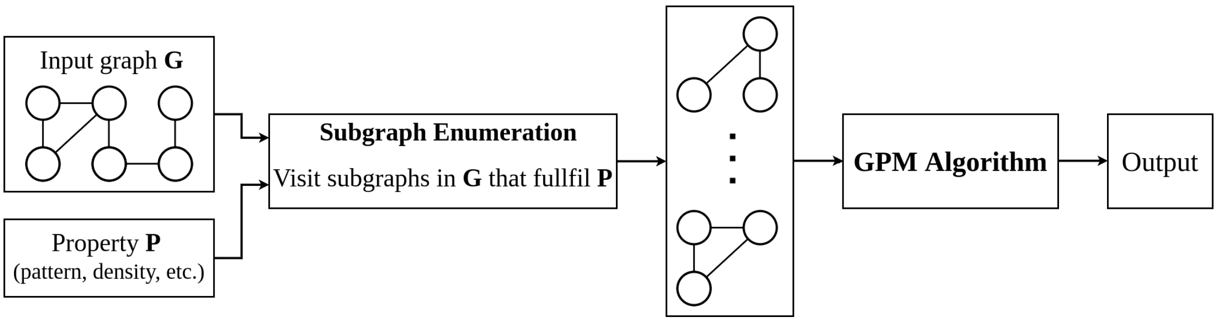
Source: [62].

(c) Collaboration detection.



Source: [18].

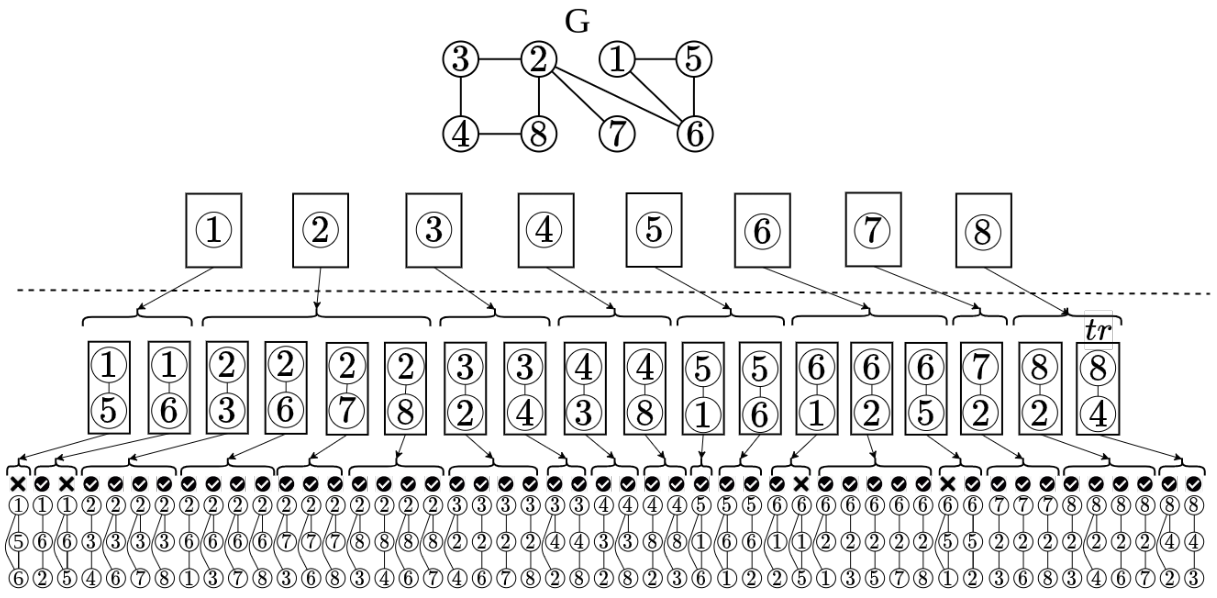
Figure 1.2: Subgraph enumeration task.



Source: created by the author.

procedure, and any traversal *tr* produces new traversals by recursively combining its vertices with a set of vertices derived from the adjacency of vertices in *tr*. For example, the traversal $tr = \{8, 4\}$ (second level) uses the vertices 2 (adjacent to 8) and 3 (adjacent to 4) to produce traversals for the third level ($\{8, 4, 2\}$ and $\{8, 4, 3\}$). A filtering in the traversals of the third level allows the extraction of subgraphs with three vertices that fullfil the desired property (paths).

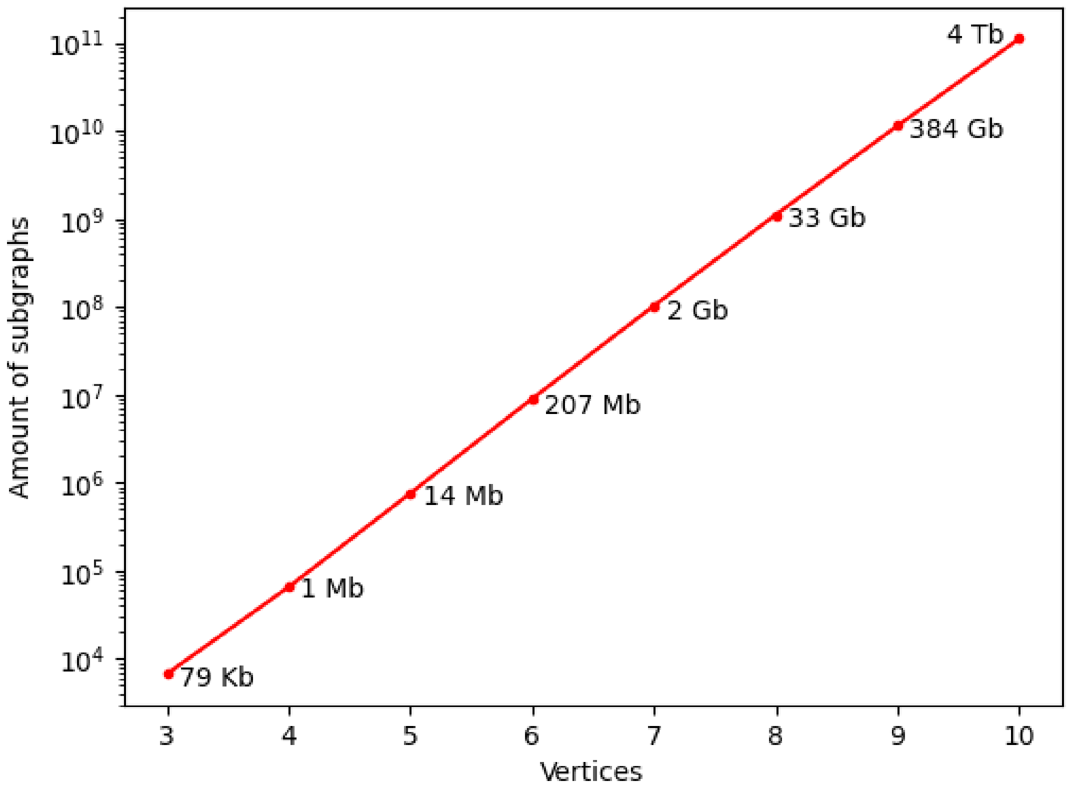
Figure 1.3: Example of subgraph enumeration lattice.



Source: created by the author.

Due to the inherent combinatorial operations needed to generate subgraphs, subgraph enumeration deals with a combinatorial explosion in the number of visited subgraphs as the size of the enumerated subgraphs increases. Figure 1.4 uses the small biological dataset *bio-diseasome* (516 vertices, 1.2K edges) [48] to depict the combinatorial explosion and the memory demand as we increase the number of vertices of enumerated subgraphs (assuming a 4-byte integer per vertex to store each subgraph). The vast amount of visited subgraphs during subgraph enumeration may cause long execution times and memory consumption, leading to the pursuit of massively parallel architectures.

Figure 1.4: Amount of subgraphs for *bio-diseasome* [48] dataset.



Source: created by the author.

Subgraph enumeration systems propose a high-level framework to allow the implementation of GPM algorithms through subgraph enumeration, providing a good trade-off between programmability and performance. These systems were proposed for CPUs [16, 28, 54, 40, 56] and GPUs [12, 11] architectures. The state-of-the-art subgraph enumeration systems for GPUs [12, 11] show that this architecture may accelerate subgraph enumeration. However, the methods in the literature do not fully exploit modern GPU computing power as they do not mitigate the critical challenges in using this device. Next, we detail the main challenges concerning parallel subgraph enumeration on GPUs.

1.1 Challenges

This section presents the main challenges concerning the design and implementation of subgraph enumeration on GPU. Algorithm 1 depicts the subgraph-centric parallel processing model [52], used in this thesis to model parallel processing. In this model, each parallel task receives a different initial traversal tr and parallel tasks perform subgraph enumeration starting from different traversals in parallel.

```
1 Function main():
2     $G \leftarrow$ input graph
3     $k \leftarrow$ desired size $k$
4     $P \leftarrow$ desired property
5     foreach $v \in V(G)$ do
6         $task[i] \leftarrow new\ Parallel\ Task(enumeration, G, k, P, \{v\})$
7     end
8 Function enumeration(Graph $G$, int $k$, Property $P$, Traversal $tr$):
9     return Subgraphs with $k$ vertices starting from $tr$ and matching $P$
```

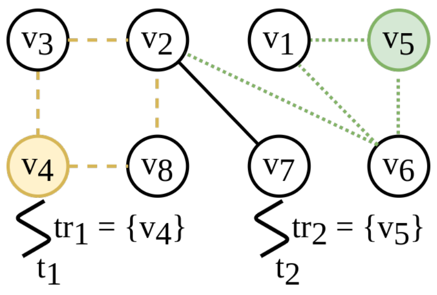
Algorithm 1: Subgraph-centric parallel processing.

1.1.1 Irregularity

Figure 1.5 depicts the parallel subgraph enumeration of two independent threads t_1 and t_2 using the same input graph of Figure 1.3. Assume t_1 and t_2 need to visit all induced subgraphs with three vertices starting from v_4 and v_5, respectively. Thread t_1 does more computations than t_2, as there are more subgraphs starting from v_4 than from v_5 (see Figure 1.3). Yellow dashed lines represent the edges accessed by t_1 through enumeration, and green dotted ones represent the edges accessed by t_2. Note that the memory access pattern of t_1 differs from t_2, as it depends on the vertices found through enumeration, which are discovered during the execution. This dynamism and unpredictability of parallel subgraph enumeration make it an irregular task, and this *irregularity* has negative consequences for efficient parallel execution on GPU.

GPUs execute threads in a *Single Instruction Multiple Data* (SIMD) fashion using groups of threads called *warps*. Threads within a warp shall execute in lockstep to take full advantage of SIMD parallelism, and the first consequence of irregularity is the degradation of lockstep execution due to *divergences* within a warp. Suppose threads t_1 and t_2 in Figure 1.5 belong to the same warp and are executing Algorithms 2 and 3 to perform

Figure 1.5: Irregularity.



Source: created by the author.

subgraph enumeration in parallel. Enumeration relies on generating the set of *extensions* (line 5), which are the vertices used to generate new traversals from the current one. In order to generate this set, threads iterate over the adjacency lists of the vertices in the traversal (lines 2 and 3) and insert into the extensions the ones that are valid according to the desired property (lines 4 and 5). Although threads t_1 and t_2 execute the same code, they iterate over adjacency lists with different sizes and progress at different paces within the same warp. This behavior generates divergent executions that decrease GPU's parallel efficiency.

```
1 Function E(traversal $tr_1$):
2     foreach $v \in tr_1$ do
3         foreach $n \in adj(v)$ do
4             if $valid(tr_1, n)$ then
5                 $tr_1.extensions+ = n$;
6             end
7         end
8     end
9     ...
```

Algorithm 2: Thread 1 enumerates tr_1.

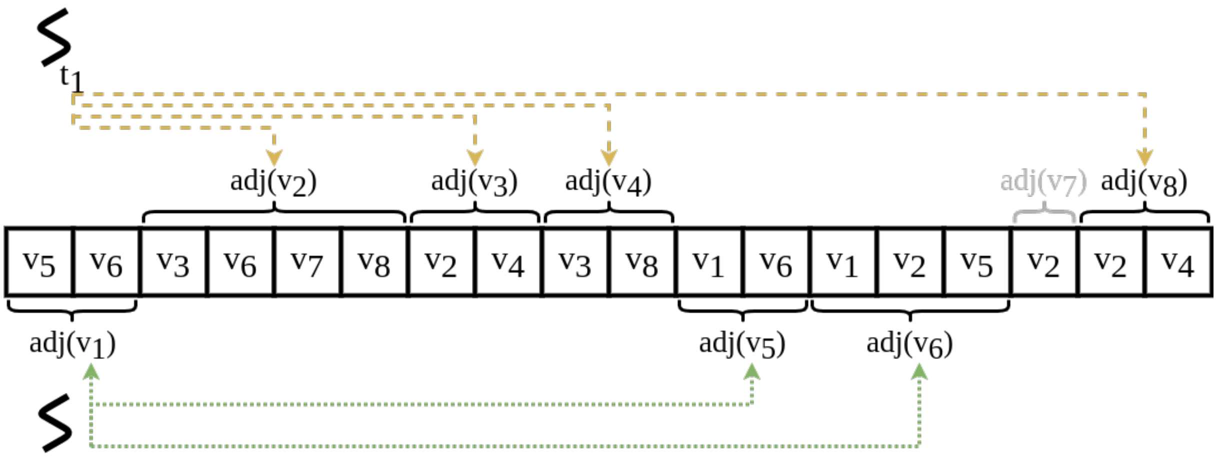
```
1 Function E(traversal $tr_2$):
2     foreach $v \in tr_2$ do
3         foreach $n \in adj(v)$ do
4             if $valid(tr_2, n)$ then
5                 $tr_2.extensions+ = n$;
6             end
7         end
8     end
9     ...
```

Algorithm 3: Thread 2 enumerates tr_2.

The second negative consequence of irregularity for GPU processing is the *memory uncoalescence*. Threads iterate over the adjacency lists to generate the extensions (line 3 of Algorithms 2 and 3), and the location of these lists in memory is dynamic throughout execution, as it depends on the vertices in the traversal. Assuming the graph is represented in CSR format [30], Figure 1.6 depicts the memory demands of threads t_1 and t_2 of Figure 1.5 to visit all induced subgraphs with three vertices. This access pattern is scattered, and t_1 and t_2 belong to the same warp, resulting in uncoalesced memory requests and the underutilization of GPU's memory bandwidth.

The GPU is a set of streaming multiprocessors (SM), each of them containing schedulers to execute warps in streaming processors (SP) in a SIMD fashion. The third negative consequence of irregularity for GPU processing is the inherent *load imbalance*, depicted by Figure 1.7. For the sake of simplicity, assume warps with two threads. In this example, the GPU has two SMs, each with two SPs, and will execute subgraph enumeration to visit all subgraphs with three vertices of the graph depicted in Figure 1.5.

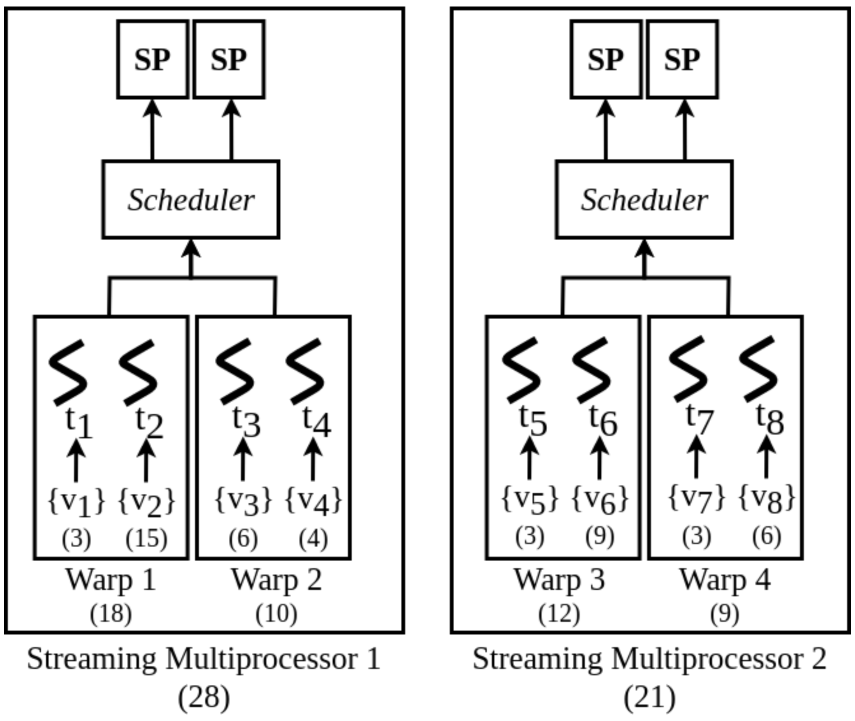
Figure 1.6: Memory uncoalescence during the access of adjacency lists.



Source: created by the author.

Each thread receives one vertex as the starting traversal, and the scheduler chooses one warp at a time to execute in the SPs from a pool of warps. The number of subgraphs visited by threads, warps, and SMs are indicated in parentheses and show the weight of the tasks processed by each SM.

Figure 1.7: Load imbalance among warps and SMs.



Source: created by the author.

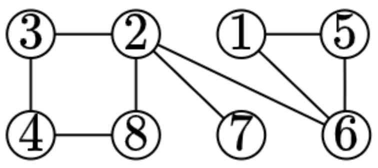
Regarding workload, there is a load imbalance among all threads within a warp. Thus, one thread will become idle earlier than the others in all warps, decreasing the warp efficiency. Besides, when we compare the number of subgraphs visited by different warps and SMs, we notice that the intra-warp load imbalance scales to the entire GPU execution. This behavior is inherent to subgraph enumeration and is enhanced by the scale-free property of real-world graphs [4], decreasing the GPU occupancy and resulting in the underutilization of GPU's massive parallelism.

1.1.2 Combinatorial Explosion

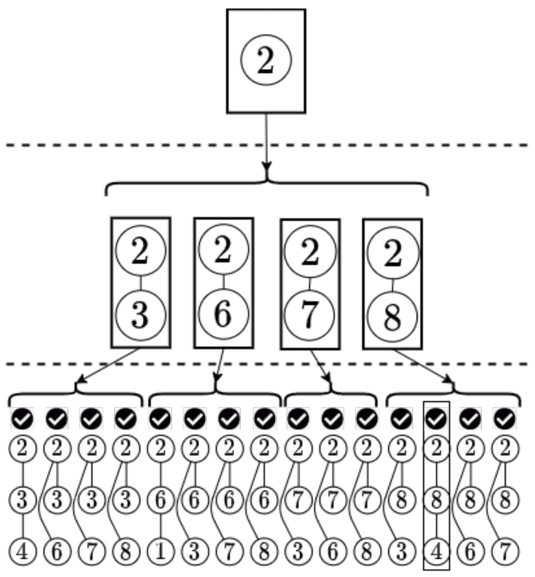
Breadth-First Search (BFS) and *Depth-First Search* (DFS) are the two standard strategies used to traverse graphs in subgraph enumeration. Figure 1.8 depicts the intermediate traversals needed by BFS and DFS strategies to generate the traversal $\{2, 8, 4\}$ using the same input graph G of Figure 1.3. BFS is a natural choice for subgraph enumeration as it exports a regular parallelism and memory locality in exploring adjacency lists. For example, the traversals in the second enumeration level of Figure 1.8(b) are produced by accessing the entire adjacency list of vertex 2 at once. However, BFS materializes all the traversals throughout enumeration, and the amount of memory required quickly grows with the size of the traversal due to the combinatorial explosion in the number of subgraphs, limiting its use to enumerate small subgraphs [12, 54].

Figure 1.8: Traversal strategies.

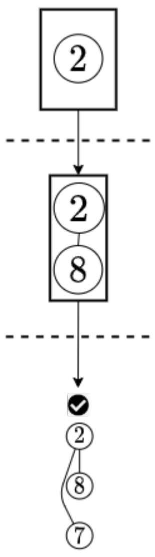
(a) Input graph



(b) BFS



(c) DFS



Source: created by the author.

The DFS approach reduces the memory demand as only a tiny portion of the states (traversals being actively processed) are kept during the enumeration. However, its irregular and strided memory requests may severely affect its parallel performance on GPU. For example, to produce the traversal $\{2, 8\}$ in the second enumeration level of Figure 1.8(c), DFS accesses only one vertex in the adjacency list of vertex 2, reducing the opportunities to take advantage of GPU's high-bandwidth memory. The strided memory access pattern of DFS hardens memory locality and deteriorates cache performance.

In summary, the combinatorial explosion in the number of traversals generates a tradeoff between memory efficiency and high memory demand concerning the traversal strategy. BFS presents better memory locality, but combinatorial explosion limits BFS for subgraph enumeration. On the other hand, DFS alleviates the effects of the combinatorial explosion at the cost of a worse memory locality.

1.2 Limitations of the State-of-the-art Strategies

As we increase the size of the subgraphs visited by subgraph enumeration, we may extend and improve the results produced by the GPM algorithms, as observed in example applications presented below:

- Network motif algorithms are used to discover relevant patterns in collaboration networks, and new collaboration patterns are found when the size of the visited subgraphs is increased up to 5 vertices [13];

- Large cliques (up to 10 vertices) are useful to extract relevant hierarchical relations in several real-world datasets [42]. Besides, algorithms that enumerate large quasi-cliques (a relaxation of the clique problem that returns only subgraphs fulfilling a density criterion) are helpful to detect representative human protein-protein interaction in biological networks [6];

- Graph compression algorithms [32, 49] may use the frequency of subgraphs to reduce the graph size in disk, usually by replacing frequent subgraphs with a shorter representation. The larger are the subgraphs visited by subgraph enumeration, the higher are the chances to find larger frequent patterns, thus improving the quality of compression.

The GPU subgraph enumeration systems in the literature do not fully exploit modern GPU computing power as they do not fully mitigate the critical challenges in using this device for subgraph enumeration: *irregularity* and *combinatorial explosion*.

Consequently, they have scalability issues concerning the amount and the size of the visited subgraphs.

Pangolin [12] and *G2Miner* [11] are the two most relevant state-of-the-art subgraph enumeration systems designed for GPUs. Pangolin follows the pattern-oblivious paradigm and uses the BFS exploration strategy. The BFS strategy generates high memory demand due to the combinatorial explosion. As a pattern-oblivious approach, Pangolin needs to perform isomorphism tests to filter subgraphs matching specific patterns. Pangolin performs these tests on the CPU, generating a performance bottleneck. Besides, Pangolin does not leverage optimizations to mitigate the impacts of irregularity during parallel processing of subgraph enumeration on GPU (divergences, uncoalescence, and load imbalance).

G2Miner [11] follows the pattern-aware paradigm and uses a DFS-like exploration strategy. As a pattern-aware approach, G2Miner needs to generate custom exploration plans to visit subgraphs matching a property. G2Miner provides a code generator to create custom GPU kernels for patterns representing a desired property, but it does not generate functional GPU codes for new patterns. Consequently, G2Miner is limited by the input patterns available in the source code and does not propose strategies to deal with the combinatorial explosion in the amount of patterns as we increase the size of the desired patterns. For example, the motif counting application searches all possible patterns with k vertices, and G2Miner mines patterns up to 4 vertices. Even in the clique application, G2Miner is limited to enumerate subgraphs up to 8 vertices.

1.3 Thesis Statement

The central hypothesis of this thesis is that the design of novel subgraph enumeration strategies for GPUs to provide more regular execution with reduced memory consumption improves the efficient use of GPU's computing power for this problem. To show that, we design novel subgraph enumeration strategies for GPUs, implement them in a GPU-friendly system, and perform an extensive evaluation using real-world datasets.

1.4 Contributions

The specific contributions of this thesis include:

- A novel traversal strategy designed for GPUs called *DFS-wide*, that provides the benefits of both BFS and DFS strategies. It presents a reduced memory demand by alternating between BFS and DFS phases, storing intermediate states with good memory locality. While the memory consumption of BFS is exponential due to the materialization of all intermediate states, DFS-wide presents a quadratic space complexity w.r.t. the size of visited subgraphs;

- A subgraph enumeration workflow designed using *regular enumeration phases*. We use the warp-centric model [24] to create enumeration steps with reduced divergences and improved memory efficiency, taking advantage of DFS-wide's memory locality to perform coalesced requests. The use of DFS-wide along with the warp-centric steps provides speedups up to 26× w.r.t. the baseline DFS version;

- A *load-balancing layer* that mitigates the inherent load imbalance of subgraph enumeration with minimum overhead to the GPU execution. Our strategy uses the CPU to monitor GPU occupancy asynchronously and performs workload redistribution. We use fine-grained information about the warps' activity to decide when GPU should be rebalanced, and redistribute enumeration tasks according to the weight of the warps to provide a balanced redistribution. The inclusion of the load-balancing layer in the warp-centric version provides speedups up to 99× w.r.t to the warp-centric version without load-balancing;

- A subgraph enumeration system named *DuMato*, which implements our optimized strategies to execute GPM algorithms efficiently on GPUs. DuMato proposes a GPU-friendly implementation of the pattern-oblivious paradigm, and is not bounded by the number of patterns and the size of subgraphs enumerated. We propose a novel strategy for canonical relabeling which allows subgraph isomorphism tests inside GPU, eliminating the overhead of calling CPU frameworks such as Nauty [41]. DuMato is able to scale for larger subgraphs and achieves speedups up to two orders of magnitude w.r.t. state-of-the-art subgraph enumeration systems. We have made our system publicly available through the following link: https://github.com/samuelbferraz/DuMat

1.4.1 Publications

This thesis has generated the following publication and submission, respectively:

- **FERRAZ, SAMUEL**; DIAS, VINICIUS; TEIXEIRA, CARLOS H. C.; TEODORO, GEORGE; MEIRA, WAGNER. Efficient Strategies for Graph Pattern Mining Algorithms on GPUs (published on IEEE SBAC-PAD 2022).
- **FERRAZ, SAMUEL**; DIAS, VINICIUS; TEIXEIRA, CARLOS H. C.; PARTHASARATHY, SRINIVASAN; TEODORO, GEORGE; MEIRA, WAGNER. DuMato: An Efficient Warp-Centric Subgraph Enumeration System for GPU (submitted to JPDC 2023, under review).

We also contributed in the following publication:

- DIAS, VINICIUS; **FERRAZ, SAMUEL**; VADLAMANI, ADITYA; ERFANIAN, MAHDI; TEIXEIRA, CARLOS H. C.; GUEDES, DORGIVAL; MEIRA, WAGNER; PARTHASARATHY, SRINIVASAN. Graph Pattern Mining Paradigms: Consolidation and Renewed Bearing (accepted on IEEE HiPC 2023).

1.5 Organization

This thesis is organized as follows:

- **Chapter 2 - Background**: In this chapter, we present the concepts concerning graph pattern mining and the foundations of the GPU architecture;
- **Chapter 3 - Related Work**: In this chapter we review the literature of subgraph enumeration systems designed for GPUs and CPUs;
- **Chapter 4 - DuMato Subgraph Enumeration System for GPUs**: In this chapter we introduce our system that is used as a software platform to develop our strategies for subgraph enumeration on GPUs;
- **Chapter 5 - Strategies for Efficient Enumeration on GPUs**: In this chapter, we present our strategies for efficient subgraph enumeration on GPUs using DuMato;

- **Chapter 6 - Experimental Evaluation**: In this chapter we provide an experimental evaluation to show the improvements in our strategies and compare our optimal implementation to the state-of-the-art subgraph enumeration systems;
- **Chapter 7 - Final Remarks**: In this chapter, we list the conclusions of this thesis and propose extensions to address some limitations.

Chapter 2

Background

This chapter presents the main definitions for designing and implementing our strategies for efficient subgraph enumeration on GPU. Section 2.1 presents the pertinent graph theory concepts, the formal definition for subgraph enumeration, and examples of GPM algorithms modeled according to this definition. Section 2.2 presents architectural details of GPUs that are important to understanding the effectiveness of our proposed strategies.

2.1 Graph Pattern Mining

2.1.1 Graph Theory

For the sake of simplicity, we assume connected undirected graphs without labels (Definitions 1 and 2). Despite that, the solutions proposed throughout this thesis may be adapted to support directed graphs and labeled components.

Definition 1 *A **graph** G is a structure defined by two sets: vertices and edges, denoted as $V(G)$ and $E(G)$, respectively. If G has n vertices, $V(G)$ is a set $\{v_1, v_2, ..., v_n\}$ and $E(G)$ is a set of unordered pairs (v_i, v_j) such that $v_i \in V(G)$, $v_j \in V(G)$, $v_i \neq v_j$ (no self-loops) and v_i is connected to v_j.*

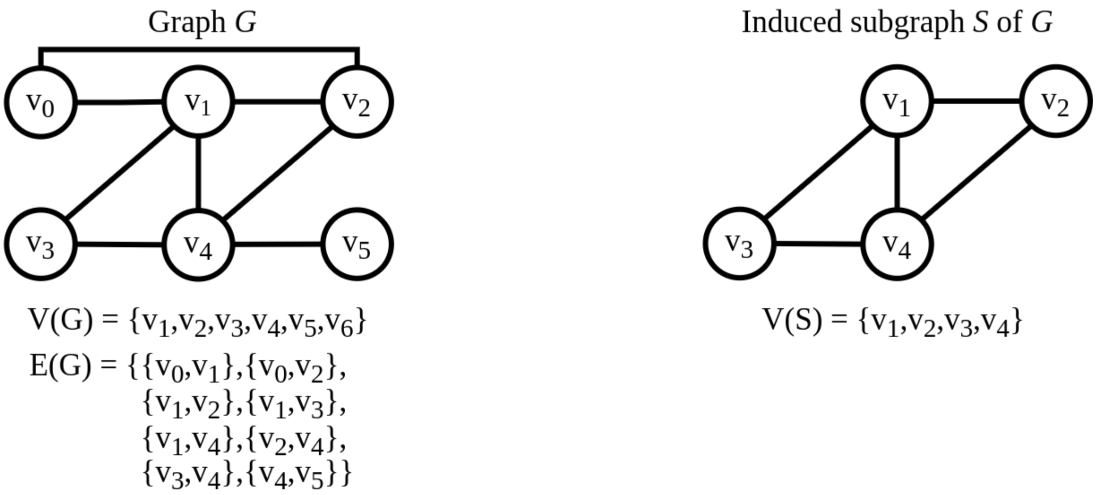
Definition 2 *A graph S is a **subgraph** of a graph G iff $V(S) \subseteq V(G)$ and $E(S) \subseteq E(G)$. S is **connected** iff for every pair of vertices $(v_i, v_j) \in V(S)$, v_i reaches v_j using edges in $E(S)$.*

We say a subgraph S of a graph G is an *induced subgraph* (Definition 4) if $E(S)$ contains all edges connecting $V(S)$ in G. Without loss of generality, in this work we assume induced subgraphs that are connected (Definition 3). Thus, if not otherwise specified, when we say *subgraph* we mean *connected induced subgraph*. Figure 2.1 depicts a graph G and an induced subgraph S of G.

Definition 3 *A* ***connected subgraph*** *of a graph G is a subgraph S such that, for any two vertices $v_i, v_j \in V(S)$, there is a path connecting v_i to v_j in S.*

Definition 4 *An* ***induced subgraph*** *of a graph G is a subgraph S such that for any two vertices $v_i, v_j \in V(S)$, $(v_i, v_j) \in E(S)$ iff $(v_i, v_j) \in E(G)$. Induced subgraphs are usually represented only by their vertex set because edges are implicitly drawn from G.*

Figure 2.1: A graph G and an induced subgraph S.



Source: created by the author.

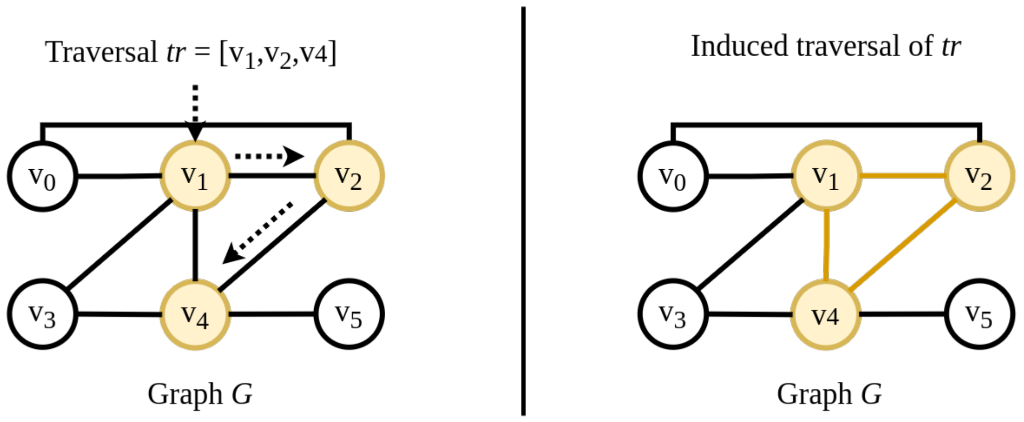
Subgraphs are discovered through incremental visits of vertices, called *traversals* (Definition 5). A traversal represents an order that vertices are visited in a graph. Any traversal is extracted from a set called *traversal power set* (Definition 6). When a traversal is finished, it can be used to create an induced subgraph from its vertices, which is called *induced traversal* (Definition 7). Figure 2.2 depicts a traversal *tr* and an induced traversal using an input graph G.

Definition 5 *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 $v \in tr$ was visited in G (first appearance). Given any two vertices $x, y \in tr$ such that $tr[i] = x$, $tr[j] = y$ and $0 \leq i < j < k$, there is a path between x and y in G, and tr visited x prior to y.*

Definition 6 *Given a graph G, the* ***traversal power set*** *(TPS) is the set of all valid traversals in G.*

Definition 7 *Given a traversal T in a graph G, an* ***induced traversal*** *is the induced subgraph S such that $V(S) = T$. We also say T induces S.*

Figure 2.2: A traversal and an induced traversal.



Source: created by the author.

A traversal's neighborhood (Definition 8) corresponds to the set containing the unique vertices in the adjacency lists of vertices in the traversal. Algorithm 4 depicts a pseudocode to determine the neighborhood of a traversal. A traversal uses its neighborhood to visit new subgraphs.

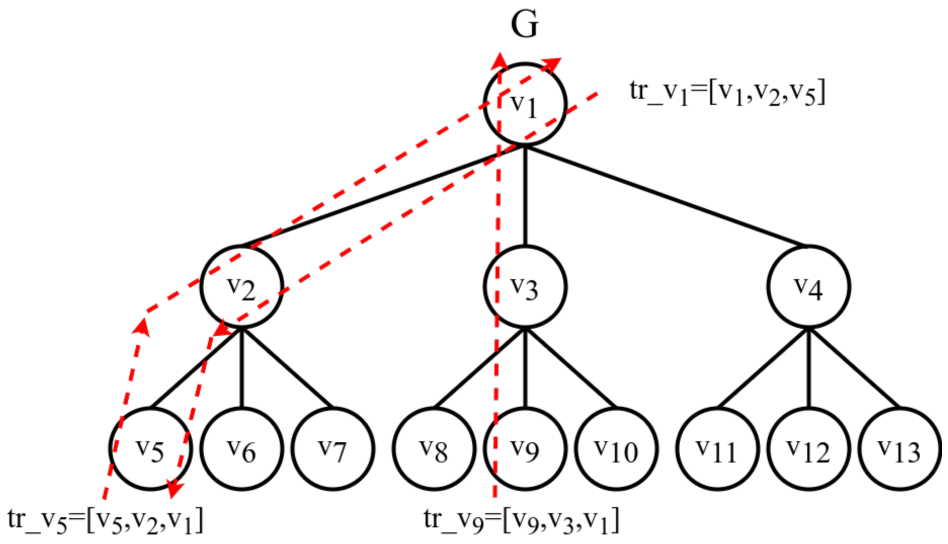
Definition 8 *Given a graph G and a subgraph S of G, the **neighbourhood** of S is a function $N : V(S) \to V(G)$ such that $N(V(S)) = \left(\bigcup_{u \in V(S)} neighbours(u) \right) \setminus V(S)$.*

```
Function N(tr):
    $neighbourhood \leftarrow \emptyset$
    foreach $v \in tr$ do
        foreach $n \in adjacency(v)$ do
            $valid \leftarrow true$
            foreach $v \in tr$ do
                if $n == v$ then
                    $valid \leftarrow false$
                    $break$
                end
            end
            if $valid$ then
                $neighbourhood \leftarrow neighbourhood \cup n$
            end
        end
    end
    return $neighbourhood$
```

Algorithm 4: Pseudocode to calculate the neighbourhood of a traversal tr.

Consider the graph G depicted in Figure 2.3 and the three traversals with 3 vertices starting from vertices v_1 (tr_v_1), v_5 (tr_v_5) and v_9 (tr_v_9). These three traversals visit subgraphs that correspond to a path, and we say there is an *isomorphism* (Definition 9) between them. Traversals tr_v_1 and tr_v_5 are not only isomorphic but also correspond to the same subgraph in G, and we say there is an *automorphism* between them (Definition 10).

Figure 2.3: Traversals tr_v_1, tr_v_5 and tr_v_9 are isomorphic. Traversals tr_v_1 and tr_v_5 are automorphic.



Source: created by the author.

Definition 9 *An* ***isomorphism*** *between two graphs G and H is a bijective function $f : V(G) \to V(H)$ such that, for all edges $(v_i, v_j) \in E(G)$, $(f(v_i), f(v_j)) \in E(H)$.*

Definition 10 *Given graphs G and H such that $V(G) = V(H)$, we say an isomorphism between G and H is an* ***automorphism****.*

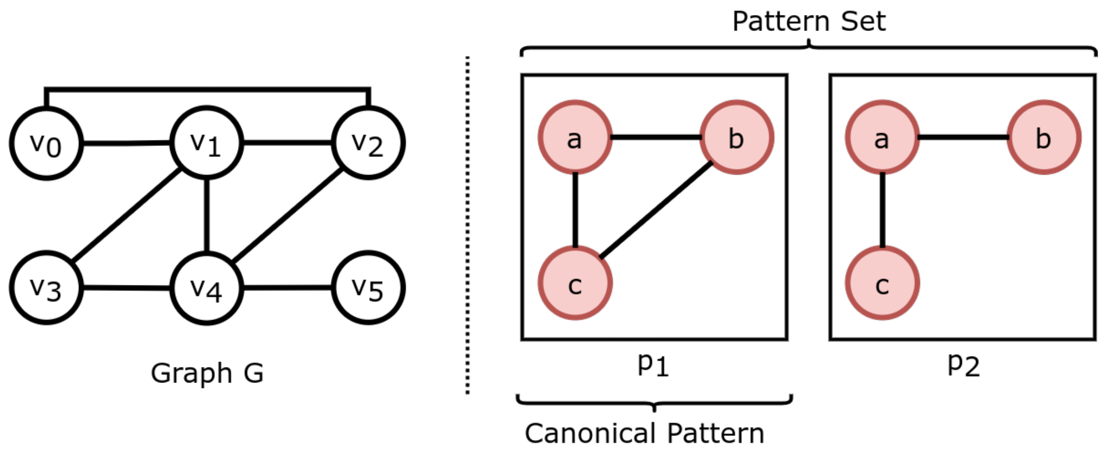
Consider now the graph G depicted in Figure 2.4. Given any induced traversal S with 3 vertices obtained from G, there must be an isomorphism between S and one of the subgraphs depicted in red. The set of subgraphs depicted in red is called *pattern set* (Definition 11), and subgraphs p_1 and p_2 are called *canonical patterns*.

Definition 11 *The* ***pattern set*** *(PS) is the minimal set of induced subgraphs (patterns) with k vertices such that, given any graph G and an induced subgraph s of G with k vertices, $\exists\ p \in PS$ such that s is isomorphic to p. Patterns in PS are also called* ***canonical patterns****, as they are stored using a standard and unique representation.*

Canonical relabeling (Definition 12) is a common task in GPM algorithms and maps a subgraph s to its corresponding canonical pattern.

Definition 12 *Given an input graph G, a subgraph s of G with k vertices and the corresponding pattern set, the* ***canonical relabeling*** *task consists in discovering which pattern*

Figure 2.4: A graph G and its possible subgraphs (patterns) with three vertices.



Source: created by the author.

in the pattern set (PS) is isomorphic to s and returning the corresponding isomorphism function.

Next we present the problem definition, which is used to design and implement Graph Pattern Mining algorithms in this thesis.

2.1.2 Problem Definition

Graph Pattern Mining (GPM) algorithms aim to generate subgraphs with k vertices that satisfy a desired property. Definition 13 presents subgraph enumeration using a function E, which reaches traversals with k vertices starting from an initial traversal. Function P models the property that enumerated subgraphs are expected to fulfill. GPM algorithms are expected to generate an output (e.g., counting the number of subgraphs), and function A receives each enumerated subgraph with k vertices to produce the expected output of the algorithm.

Definition 13 *$E : (G, tr, k, P, A) \rightarrow TR$ is a function such that:*

$$E(G, tr, k, P, A) = \begin{cases} \bigcup\limits_{u \in N(tr)} E(G, P(tr+u), k, P, A) & \text{if } 0 < |tr| < k \quad (1) \\ A(tr) & \text{if } |tr| = k \quad (2) \\ \emptyset & \text{if } |tr| = 0 \quad (3) \end{cases}$$

where G is a graph; $tr \in TPS$; $k \geq |tr|$; $P : TPS \rightarrow TPS$ is a property such that $P(tr) = tr$ if tr satisfies a property, and $\emptyset$ otherwise; $A(tr)$ produces results for the GPM algorithm; and $TR \subseteq TPS$.

In order to implement E, we must choose a traversal strategy to visit subgraphs. The two main traversal strategies are **breadth-first search** (*BFS*) and **depth-first search** (*DFS*), and Algorithm 5 depicts implementations of a function E using these strategies. BFS builds a queue to materialize all traversals generated throughout enumeration. On the other hand, DFS materializes only one traversal at a time and extends it recursively.

```
Function E_BFS($G$, $tr$, $k$, $P$, $A$):
    $queue \leftarrow \{tr\}$
    while $queue \neq \emptyset$ do
        $tr \leftarrow dequeue(queue)$
        if $|tr| == 0$ then
            return
        end
        else if $|tr| == k$ then
            $A(tr)$
        end
        else
            foreach $v \in N(tr)$ do
                $tr' \leftarrow tr + v$
                $enqueue(queue, P(tr'))$
            end
        end
    end

Function E_DFS($G$, $tr$, $k$, $P$, $A$):
    if $|tr| == 0$ then
        return
    end
    else if $|tr| == k$ then
        $A(tr)$
    end
    else
        foreach $v \in N(tr)$ do
            $tr' \leftarrow tr + v$
            $E\_DFS(P(tr'), k)$
        end
    end
```

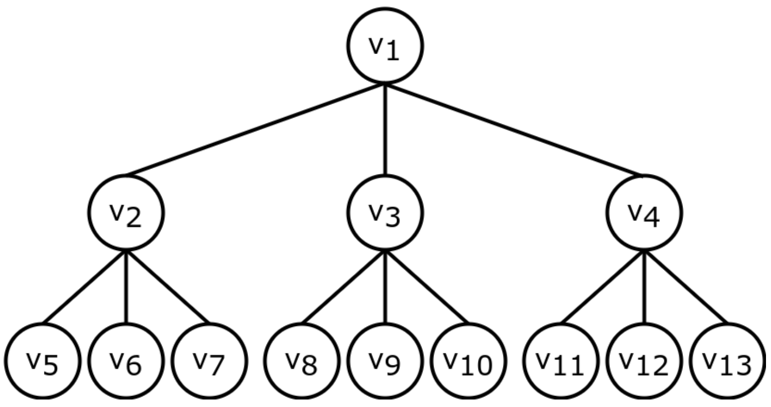
Algorithm 5: Algorithms for function E using BFS and DFS traversals.

Given the graph G of Figure 2.5(a), consider the calls E_BFS and E_DFS depicted in Figures 2.5(b) and 2.5(c). Assume a function P that allows visiting all possible traversals. The green induced traversals to represent the first occurrence of the same induced traversal $[v_1, v_2, v_5]$ in both exploration strategies. BFS visits subgraphs $[v_1, v_3]$ and $[v_1, v_4]$ (represented in yellow), keeps these two intermediate states in memory, and reaches $[v_1, v_2, v_5]$ afterward. On the other hand, DFS goes straight to $[v_1, v_2, v_5]$ without keeping intermediate states $[v_1, v_3]$ and $[v_3, v_4]$.

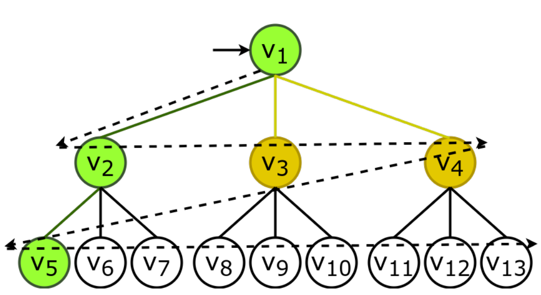
In order to understand how the enumeration function E may be used to design GPM algorithms, consider the GPM algorithms *clique listing* (Definition 14) and *motif counting* (Definition 15), depicted by Figure 2.6. Algorithms 6 and 7 depict the design of GPM algorithms for these problems. Without loss of generality, we choose the DFS implementation of enumeration function E.

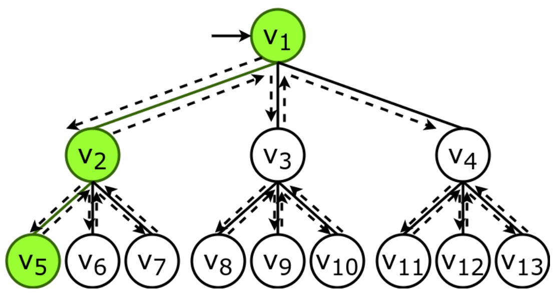
Definition 14 *A **clique** of size k is a graph C with k vertices such that, for every $v_i \in V(C)$ and $v_j \in V(C)$, $(v_i, v_j) \in E(C)$. Given a graph G, the **clique listing** problem lists all cliques of G with k vertices.*

Figure 2.5: Traversal strategies



(a) Graph G

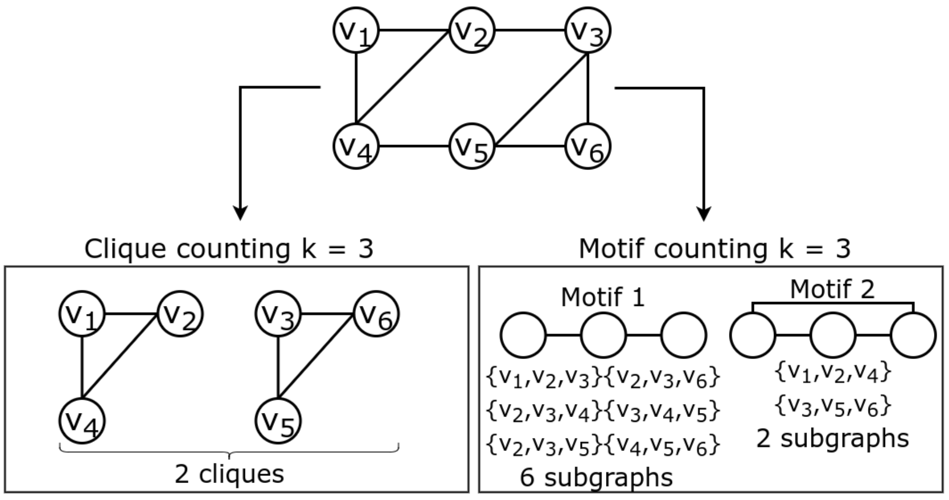




(b) $E\_BFS(G,\ \{v_1\},\ k,\ P,\ A)$ (c) $E\_DFS(G,\ \{v_1\},\ k,\ P,\ A)$

Source: created by the author.

Figure 2.6: Clique counting and motif counting algorithms.



Source: created by the author.

Definition 15 *A* ***motif*** *of size k is a vertex-induced connected subgraph containing k vertices. The* ***motif counting*** *problem counts the amount of each motif of size k in a graph G.*

For clique listing (Algorithm 6), property function P checks whether the induced traversal is a clique, and A function prints the cliques with k vertices. For motif counting

```
Function P($tr$):
    foreach $v \in tr$ do
        foreach $v' \in tr$ do
            if $v \neq v'$ and $v' \notin adj(v)$ then
                return $\emptyset$
            end
        end
    end
    return $tr$

Function A($tr$):
    $print(tr)$

Function clique_listing($G$, $k$):
    foreach $v \in V(G)$ do
        $E\_DFS(G, v, k, P, A)$
    end
```

Algorithm 6: Clique listing using enumeration function E.

```
Function P($tr$):
    return $tr$

Function A($tr$):
    $s \leftarrow induced\_traversal(tr)$
    $pattern \leftarrow canonical\_relabeling(s)$
    $count[pattern] \leftarrow count[pattern] + 1$

Function motif_counting($G$, $k$):
    foreach $v \in V(G)$ do
        $E\_DFS(G, v, k, P, A)$
    end
```

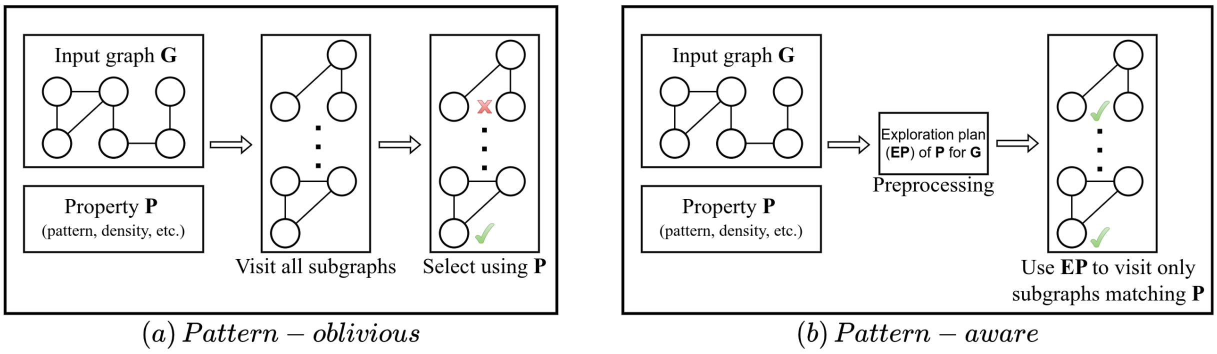
Algorithm 7: Motif counting using enumeration function E.

(Algorithm 7), property function P allows all traversals to pass (any induced traversal with k vertices is valid), and function A discovers which canonical pattern p is isomorphic to the traversal and increments the counter associated with p.

2.1.3 Subgraph Enumeration Paradigms

Given a graph G and a property P, Figure 2.7 depicts the two subgraph enumeration paradigms used to visit subgraphs matching P in G: *pattern-oblivious* and *pattern-aware*. In the pattern-oblivious approach, all subgraphs of a specific size are visited regardless of P, and the ones that fulfill P are selected afterwards. In order to filter subgraphs matching a pattern, this approach relies on subgraph isomorphism tests. Unfortunately, the state-of-the-art subgraph isomorphism tools are designed for CPU [41, 29]. Besides, visiting subgraphs regardless of the property increases the number of visited subgraphs and the overall cost of this approach.

Figure 2.7: Subgraph enumeration paradigms.



Source: created by the author.

The pattern-aware approach uses the desired graph property to create an exploration plan [7, 23] to visit only subgraphs matching the property. Instead of materializing all subgraphs and checking properties afterward, pattern-aware uses the exploration plan to extract from the adjacency lists only the extensions that produce new subgraphs matching the property, eliminating the need of graph isomorphism algorithms. The main disadvantage of pattern-aware is the need of custom rules for each pattern associated with a property. For example, if you use this paradigm to visit quasi-cliques, you will have to create custom rules for each possible pattern associated with the desired quasi-cliques. When we increase the size and the amount of patterns, the cost and complexity of generating thousands of exploration plans may limits the scalability of subgraph enumeration systems using this paradigm. For example, G2Miner [11] has a hardcoded restriction that limits the execution of the motif counting application (which visits all subgraphs with a certain size) for subgraphs up to 4 vertices.

Next we present the main concepts concerning our target computing architecture: GPUs.

2.2 GPU Architecture

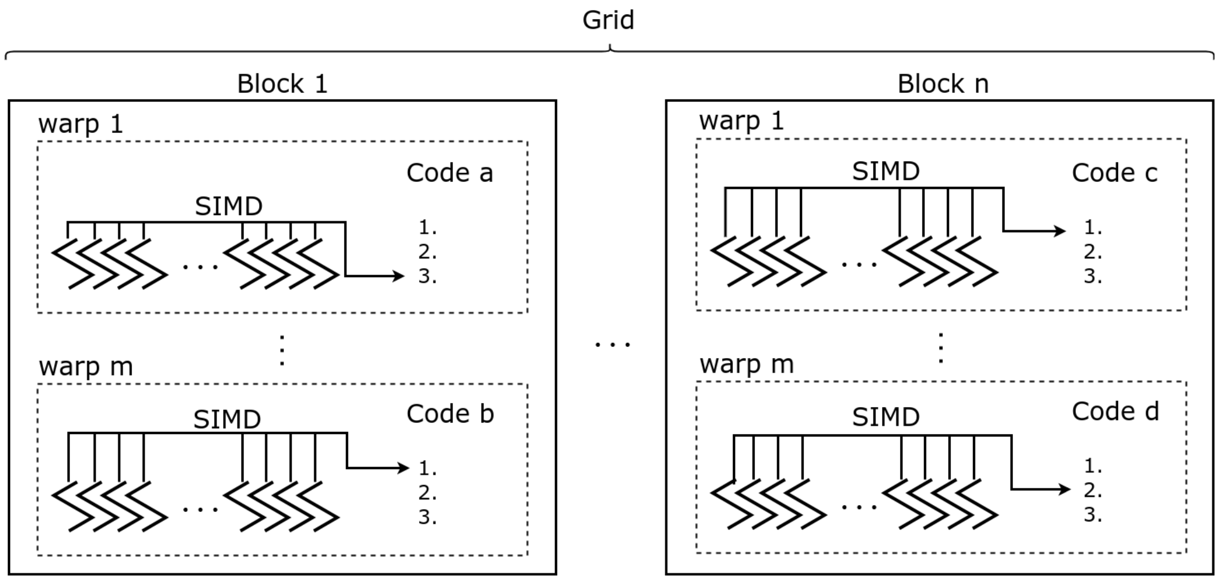
GPUs are parallel computing devices containing thousands of cores and a high-bandwidth DRAM. The two leading manufacturers of current GPUs are NVIDIA [44] and AMD [3]. Although they differ concerning low-level hardware, their execution and memory models share the same architectural concepts. NVIDIA's GPUs have been widely used to accelerate applications, and we will explain the architectural details of GPUs using NVIDIA's nomenclature.

GPUs follow the ***S**ingle **I**nstruction **M**ultiple **T**hread* (SIMT) execution model, depicted in Figure 2.8. A code is executed in a ***S**ingle **I**nstruction **M**ultiple **D**ata* (SIMD) fashion by groups of threads called *warps*. Each thread of a warp has a local sequential id called *lane*, which is an integer in the range $[0 \cdots warpsize - 1]$ used to distiguish each thread inside the warp. Threads within a warp are supposed to execute the same instruction in lockstep using their data. A warp is an independent computing unit, and different warps do not share program counters. Warps are organized in groups called *blocks*, and the *grid* is the set comprising all blocks. The warp size is fixed (usually 32) and depends on the GPU architecture, but the programmer defines the block and grid sizes. Threads can be identified concerning the grid (*global id*), to the block (*local id*), and to the warp (*private id*).

Figure 2.9 depicts the execution and memory architecture of current GPUs. A GPU is an array of ***S**treaming **M**ultiprocessors* (SM), which are responsible for executing a set of warps. Each SM receives a set of blocks to execute, and warps within the same block are scheduled to execute in the same SM. Each SM contains the following:

- *Cores* (CO), the smallest execution unit. A group of n cores is scheduled to execute a warp with n threads in a SIMD fashion. An SM usually contains 32 to 128 cores [44];
- *Register File*, a private memory of each core. The amount of registers allocated for warps depends on the total amount of threads scheduled to execute on the GPU;
- *Warp Schedulers*, which schedules the execution of warps within an SM. Each SM

Figure 2.8: SIMT execution model.



Source: created by the author.

contains its warp schedulers, as SMs are supposed to schedule warps independently. The amount of warp schedulers per SM depends on the warp size and the number of cores per SM. For example, an SM with 64 cores executing 32-thread warps contains two warp schedulers;

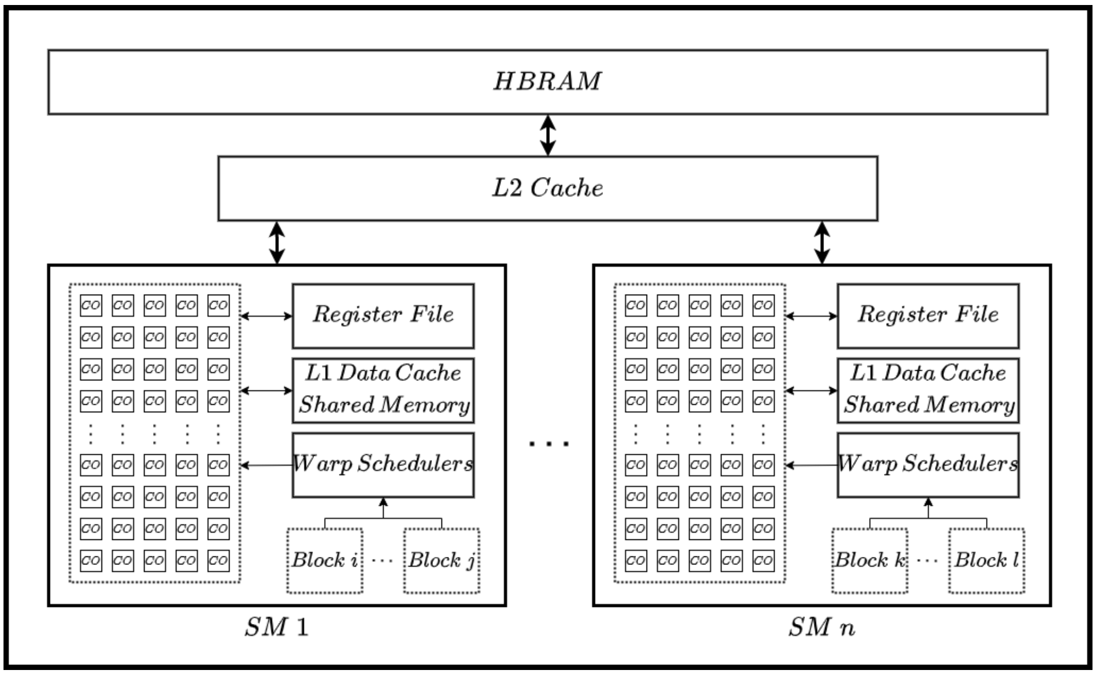
- *L1 Data Cache and Shared Memory*, a configurable memory cache shared among cores within the same SM. It can be used as an L1 hardware data cache and/or as a shared memory for warps executing in the same SM.

Memory requests not serviced by the internal SM memory hierarchy are forwarded to the L2 cache and HBRAM. The L2 cache has a fixed size and provides hardware caching with spatial and temporal locality. The HBRAM is a high bandwidth DRAM that can service parallel memory requests using few memory transactions when specific requirements are met.

The primary requirement to effectively use HBRAM is: the *k-th* thread within a warp must access the *k-th* word of the same 32-byte, 64-byte, or 128-byte memory segment. Memory requests are serviced with memory transactions that are 32-byte, 64-byte, or 128-byte wide. Fewer memory transactions are needed when threads within the same warp request positions in the same memory segment.

Algorithm 8 depicts the code of three warps w_1, w_2 and w_3 (32 threads each). The argument *vec* is an array of 4-byte integers, and the function *private_id*() returns an integer in range $[0 \cdots 31]$, corresponding to the thread id with respect to the warp. In order to understand how memory requests are serviced on GPUs, we will analyze the number of memory transactions to access *vec*[*id*] in each warp (last line of each function).

Figure 2.9: GPU's execution and memory model.



Source: created by the author.

```
1  Function $w_1$($int*\ vec$):
2      $int\ id \leftarrow private\_id()$
3      $int\ value \leftarrow vec[id]$

4  Function $w_2$($int*\ vec$):
5      $int\ id \leftarrow private\_id()$
6      $id \leftarrow id * 32$
7      $int\ value \leftarrow vec[id]$

8  Function $w_3$($int*\ vec$):
9      $int\ id \leftarrow private\_id()$
10     if $id\%2 == 0$ then
11         ...
12     $int\ value \leftarrow vec[id]$
```

Algorithm 8: Functions representing code executed by three different warps.

For w_1, thread 0 requests $vec[0]$, thread 1 requests $vec[1]$, thread 2 requests $vec[2]$ and so forth. Thus, memory requests of w_1 are within the same 128-byte memory segment ($vec[0 \cdots 32]$), and we say the GPU services these requests using only one coalesced transaction. For w_2, thread 0 requests $vec[0]$, thread 1 requests $vec[32]$, threads 2 requests $vec[64]$ and so forth. Thus, each memory request of w_2 belongs to a different 128-byte memory segment, and we say the GPU services these requests using 32 uncoalesced transactions.

Warp w_3 shows the importance of regularity for efficient memory requests. As w_1, memory requests of w_3 belong to the same 128-byte memory segment. Despite that, half of w_3's threads (odd ids) will reach line 12 prior to the other half (even ids) due to the conditional statement in line 10. We say there is a *branch divergence* in threads of w_3, as threads within the same warp need to execute different instructions after a certain execution point. After divergence, w_3 is divided into two execution units (odd and even threads), which are scheduled independently, including their memory requests. Thus, although all memory requests of w_3 belong to the same memory segment, two identical transactions ($vec[0 \cdots 32]$) are generated to service w_3 due to divergence. Half of the positions of each transaction are not used and represent wasted bandwidth.

Chapter 3

Related Work

This thesis aims to design novel strategies to improve the efficiency of graph pattern mining algorithms on GPUs. Thus, we must be able to deal with the execution behavior and memory consumption of this class of algorithms. This section reviews the literature on general-purpose subgraph enumeration systems designed for CPUs and GPUs.

Table 3.1 summarizes the works discussed. We analyze how their strategies mitigate the challenges for parallel subgraph enumeration (irregularity and combinatorial explosion). For CPU systems, we analyze regularity by inspecting memory locality and load balancing. For GPU systems, we analyze regularity by inspecting divergences, memory uncoalescence, and load-balancing strategies. For both architectures, we analyze the impacts of the combinatorial explosion phenomenon by examining the memory requirements of each system.

Parallel implementations of specific GPM algorithms were proposed for multicore [14, 1, 34, 47, 19] and manycore architectures [31, 36, 55, 26, 2]. However, these solutions do not provide a general-purpose environment that allows the design of other GPM applications and will not be covered in this chapter.

3.1 Subgraph Enumeration Systems for CPU

Arabesque [54] is one of the first systems targeting distributed memory machines and the first to support subgraph-centric parallel processing. 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. At the same time, it also employs load balancing using a round-robin strategy. It proposes a novel canonicality checking algorithm, which allows a coordination-free exploration strategy, along with a two-level aggregation mechanism that reduces the amount of isomorphism checking. Despite Arabesque's compressed data structure to

GPM System	Proc.	Explor. strate.	Enumeration paradigm
Arabesque [54]	CPU	BFS	Pattern-oblivious
NScale [46]	CPU	BFS	Pattern-oblivious
G-miner [10]	CPU	BFS	Pattern-oblivious
RStream [56]	CPU	BFS	Relational
Automine [40]	CPU	DFS	Pattern-aware
Fractal [16]	CPU	DFS	Pattern-oblivious
Pangolin [12]	CPU+GPU	BFS	Pattern-oblivious
Peregrine [28]	CPU	DFS	Pattern-aware
G-thinker [58]	CPU	BFS	Pattern-oblivious
GraphPi [50]	CPU	DFS	Pattern-aware
PBE [21]	GPU	DFS	Pattern-oblivious
Kaleido [61]	CPU	BFS	Pattern-oblivious
GraphZero [39]	CPU	DFS	Pattern-aware
SumPA [20]	CPU	DFS	Pattern-aware
G2Miner [11]	GPU	BFS/DFS	Pattern-aware

Table 3.1: Related work.

store intermediate enumeration data, its BFS-style enumeration limits the enumeration of bigger subgraphs due to the combinatorial explosion.

NScale [46] proposes a distributed graph computing environment for neighborhood-centric processing, which is more suitable for graph analytics algorithms such as PageRank. Thus, it only supports simple GPM algorithms (e.g., triangle counting). The authors propose optimizations to reduce the memory requirements by exploiting overlaps among the subgraphs on a machine. However, its neighborhood-centric design presents a high memory consumption as it requires the construction of subgraphs prior to execution. They divide the graph into partitions and use bin packing-based algorithms to create partitions, along with a distributed cache mechanism to improve memory locality. However, the dynamic behavior of subgraph enumeration limits the effectiveness of this load-balancing strategy.

G-miner [10] is a system whose primary goal is to propose a task pipeline to mitigate the load imbalance of enumeration tasks in a distributed environment. They bound the memory consumption caused by combinatorial explosion using secondary storage (disk) and also improved memory locality using local caches to avoid pulling remove vertices. They propose static and dynamic load-balancing schemes. The static scheme uses a greedy algorithm to divide the graph into partitions using graph coloring and

CC (connected components) finding algorithms. The dynamic scheme allows task stealing among workers, as an optimal static load balancing scheme is hard to predict for subgraph enumeration. Other parallel architectures may use the general-purpose task pipeline proposed by G-miner to allow static and dynamic load balancing schemes with reduced synchronization overheads. However, it is also a BFS-like enumeration algorithm. Although the authors use secondary storage to keep the intermediate states, the amount of I/O transactions and the size of the second storage impacts the performance and viability of enumerating bigger subgraphs.

Similarly to *G-miner*, *G-thinker* [58] proposes a task scheme for distributed graph pattern mining processing. They improve memory locality using a hierarchical caching scheme through three kinds of tables, reducing the I/O costs of concurrent accesses to vertices. It also improves load balancing by providing a more lightweight task scheme that removes the need for graph partitioning. It also proposes different task containers to model enumeration tasks. Both cache tables and task containers can be parameterized to provide bounded memory consumption. It also supports dynamic load balancing and fault tolerance. However, its BFS-like enumeration strategy still needs to eliminate the impacts of combinatorial explosion and presents the same limitations concerning the length of the enumerated subgraphs.

RStream [56] is an out-of-core single-machine relational system that relies on join operations to perform subgraph enumeration. The main goal of RStream is to provide a subgraph enumeration that can scale for bigger subgraphs by increasing disk capacity at the cost of suboptimal performance. Thus, their strategies optimize join operations and provide a friendly programming interface. They propose a load-balancing strategy that divides the intermediate tables produced by enumeration into chunks, which are pushed back into a job list managed by a produced-consumer algorithm. The user must build its relational phases using callback functions to allow canonical filters/redundancy elimination. After each join, tuple reshuffling is performed to provide memory locality in partitions. It incorporates canonical filtering and two-level pattern aggregation from Arabesque [54, 27]. They do not propose strategies to improve memory locality. The join-based scheme used by RStream is similar to a BFS-like enumeration algorithm. Thus it presents severe limitations caused by the intermediate states produced by the expensive join operations, causing high memory consumption and intensive I/O requests as the length of enumerated subgraphs increases.

Fractal [16] was the first distributed memory CPU-based system to use a DFS-like strategy for subgraph enumeration. The DFS exploration reduces the materialization of the intermediate states produced by enumeration, reducing the memory requirements and mitigating the impacts of combinatorial explosion. In order to mitigate load imbalance, they use job queues to propose a work-stealing mechanism such that threads in the same node steal local jobs prior to stealing from remote nodes, reducing the communication overheads. They do not propose strategies to improve memory locality.

Specific patterns may allow custom optimizations and execution plans, and *AutoMine* [40] proposes an automated code generation for GPM algorithms on CPU through set intersection/subtraction operations. It uses the pattern-aware exploration strategy, where patterns guide the subgraph enumeration by leveraging specialized execution plans. It employs scheduling of intersect/subtract operations to automate code generation for custom patterns, and optimizes code for a single pattern by calculating loop-invariant neighborhoods (prefixes) and saving them before nested loops. In the same way, *Peregrine* [28] is a parallel GPM system designed for shared memory CPU machines, which uses the same concepts of AutoMine but proposes an API that allows high-level programming of GPM algorithms. It proposes the concepts of anti-vertices and anti-edges to allow advanced structural constraints in the enumeration. As the pattern-aware approach is specialized for specific patterns, it may be too expensive in general-purpose GPM scenarios where subgraph exploration typically involves multiple patterns. Both systems use a DFS-like enumeration scheme, thus reducing the memory requirements. AutoMine mitigates neither memory locality nor load imbalance. Peregrine does not mitigate memory locality and uses a minimalist load balancing scheme to reduce the load imbalance across matching tasks, which is not enough to mitigate load imbalance.

GraphPi [50] and *GraphZero* [39] are pattern-aware systems that propose improvements for nested-loop-based (e.g., *AutoMine* [40]) implementations of GPM algorithms. *GraphPi* discusses the tradeoff concerning symmetry-breaking and the performance of subgraph matching. For instance, some rules may prune all automorphisms but generate unnecessary intermediate states. It proposes a novel strategy to break the symmetry of patterns (2-cycle based) and an efficient schedule generator that estimates the cost of symmetry-breaking rules along with the execution plan. At last, a strategy to deal with counting-only algorithms is proposed. *GraphZero* proposes optimizations for automated code generated by *AutoMine*. Its schedule generator uses a performance model to generate an optimal schedule, and, using this schedule, its symmetry-breaking rules are extracted and allow the reduction of automorphisms to one. At last, it generalizes the orientation optimization (standard in specific patterns such as cliques), where the vertices are reindexed, and pruning is improved. Both perform DFS-like enumeration and present reduced memory usage, thus mitigating the impacts of combinatorial explosion. *GraphPi* does not propose a strategy to improve memory locality and uses a minimalist work-stealing mechanism that keeps a task queue containing a set of tasks greater than a threshold (not enough to mitigate load imbalance for all datasets). *GraphZero* mitigates neither memory locality nor load imbalance.

Kaleido [61] is an out-of-core system designed to overcome the main limitations of *Arabesque*. It proposes another compact data structure to store intermediate enumeration states (an improvement over Arabesques's ODAG data structure) with an I/O layer to provide external storage. It also improves isomorphism checking by proposing filters to reduce the calls for canonical relabeling libraries. 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 it is possible to predict the size of subgraph candidates level by level, this information is not enough to accurately predict the volume of jobs generated by a subgraph. As *Kaleido* uses a BFS-like exploration strategy, it presents the same limitations concerning the length of the enumerated subgraphs due to the combinatorial explosion. They do not propose strategies to improve memory locality.

Nested-loop-based systems such as *AutoMine* and *Peregrine* rely on custom symmetry-breaking rules and schedule generation for each pattern enumerated. As patterns may share substructures, enumerating distinct patterns independently may represent redundant computation. Besides, GPM algorithms that enumerate thousands of patterns (e.g., motif counting) may suffer from high overheads. *SumPA* [20] proposes strategies to merge patterns according to their similarities to reduce redundant computation. They reduce the impacts of combinatorial explosion by using a DFS-like exploration scheme. They also improve load imbalance by reordering the graph's vertices using the degrees and providing metadata of parallel enumerations to allow work stealing. They improve memory locality by caching the edges of overused vertices and caching their data structure that stores shared substructures of patterns.

In the next section, we describe some subgraph enumeration systems for GPU.

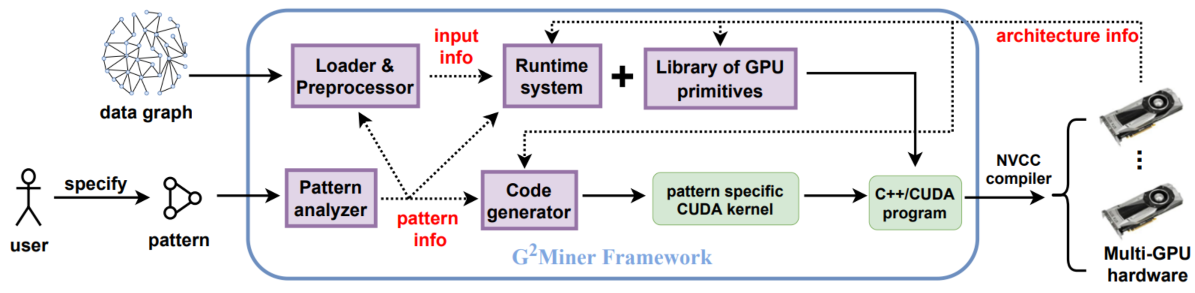
3.2 Subgraph Enumeration Systems for GPU

We describe the two main subgraph enumeration systems for GPUs: Pangolin [12] and G2Miner [11]. They were chosen because both provided a high-level API to facilitate the creation of other GPM applications and also performed an experimental evaluation of at least two distinct GPM applications. Other works such as PBE [21] and NemoGPU [36] are not described because they were designed specifically for one GPM application and do not provide a high-level programming interface to simplify the implementation of other GPM algorithms.

Pangolin [12] is a GPU system that follows the pattern-oblivious enumeration using the BFS exploration strategy. Pangolin's design enables to optimize executions by pruning the search space of subgraphs and by reducing the number of isomorphism tests required. Materialized intermediate states generated by the BFS exploration facilitate the runtime to leverage BSP (Bulk Synchronous Parallel) load-balancing schemes. Despite that, they do not propose load-balancing strategies, and the high memory demands of BFS limit its applicability to enumerate small subgraphs due to the combinatorial explosion. Besides, Pangolin does not leverage optimizations to handle memory uncoalescence and divergences of parallel GPM algorithms on GPU and relies on CPU frameworks to perform isomorphism tests.

G2Miner [11] is a pattern-aware system designed for multi-GPU processing. It allows the application to choose between a bounded BFS and a DFS enumeration, and a warp-centric execution for the set intersect operations among adjacency lists. It uses a simple work distribution scheme, which is edge-centric rather than vertex-centric. It also proposes a task-scheduling mechanism to distribute enumeration tasks among GPUs. As a pattern-aware environment, it is supposed to generate exploration plans for each query pattern, and G2Miner proposes a software architecture that generates custom GPU kernels for each pattern, as depicted by Figure 3.1. G2Miner suffers from limitations when enumerating larger subgraphs, and our experimental evaluation (detailed later in Chapter 6) reflects this. Pattern-bounded applications such as motif counting are executed only for subgraphs up to 4 vertices, and their code generator does not generate functional GPU code when a new pattern is provided as input. Besides, their load balancing scheme is too simple to mitigate the inherent load imbalance among enumeration tasks, as edge-centric workload distributions also suffers from load imbalance.

Figure 3.1: G2Miner execution workflow.



Source: [11].

The GPM systems for GPUs show that this architecture can provide performance gains concerning multicore architectures. However, these works still need to fully address the critical challenges concerning parallel subgraph enumeration on GPUs (combinatorial explosion, memory uncoalescence, divergences, and load imbalance). *Pangolin* has scalability issues due to its BFS-like exploration that materializes the massive amount of intermediate data created by the combinatorial explosion. Besides, they do not mitigate memory uncoalescence, divergences, and load imbalance. *G2Miner* provides strategies to reduce the impact of combinatorial explosion by providing a bounded BFS or DFS-like scheme, depending on the algorithm. They also provide a warp-centric scheme to perform the *set-intersection* operations and improve divergences and memory coalescence. However, the gains obtained with the warp-centric strategy are small compared to their baseline implementation (2x). For example, our warp-centric enumeration scheme achieved speedups up to 33x due to increased memory coalescence and lockstep execution (detailed later in Section 6.2). Besides, their load balancing mechanism is minimalist for the complexity of subgraph enumeration's inherent load imbalance, and their implementation of the pattern-aware subgraph enumeration paradigm does not work when a diverse set of patterns with more than 4 vertices are provided.

We propose strategies to deal with the combinatorial explosion, mitigate memory uncoalescence/divergences and load imbalance. Besides, our pattern-oblivious enumeration scheme is scalable for a more significant number of patterns compared to pattern-aware systems such as *G2Miner*.

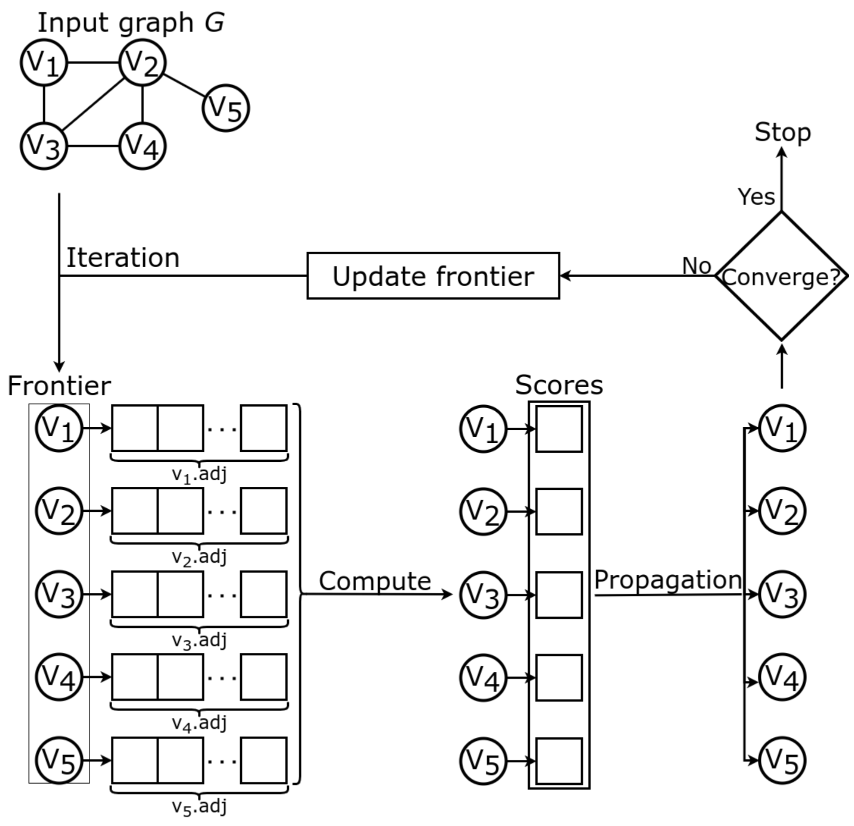
3.3 Other Graph Mining Systems

There is a similar class of graph systems primarily designed to implement graph mining algorithms such as PageRank, HITS and betweenness centrality. These algorithms have a different execution behavior and memory consumption compared to graph pattern mining ones. Figure 3.2 depicts the execution of the PageRank algorithm using GunRock's summarized workflow [57], the most popular system for GPU designed for these graph mining algorithms. Other systems such as CuSha [33] and Ligra [51] follow the same principles but using a different workflow.

The PageRank algorithm uses vertices to represent webpages and edges to represent links between them. For each page (vertex), it computes a relevance score using their incoming links (edges). The algorithm starts with a set of active vertices/edges called frontiers. In PageRank, the initial frontier is the set of all vertices. It visits the adjacency lists for each vertex in the frontier set to discover incoming links and computes the per-page scores. Once the values are calculated, a synchronization step propagates the score values to the active vertices in the frontier. These graph mining algorithms rely on convergence parameters, and, as long as a convergence criterion is not reached, they continue processing. The PageRank algorithm continues computing per-page relevance scores if the difference between the new and old scores exceeds a threshold. The algorithm updates the frontiers with the vertices whose scores should be recomputed, and the computing cycle restarts.

In order to understand the behavior of graph pattern mining algorithms, Figure 3.3 depicts the enumeration of all induced subgraphs with four vertices of the same input graph used in the PageRank example. The computation starts from each vertex, and the compute steps produce the intermediate states needed throughout enumeration.

Figure 3.2: Graph analytics execution workflow.

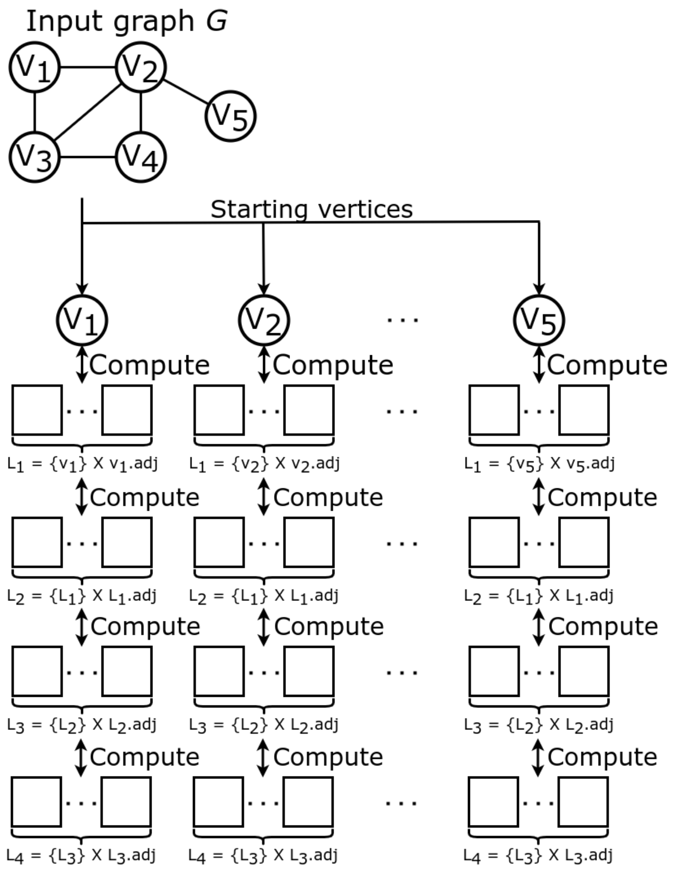


Source: created by the author.

L_i represents the *i-th* set of intermediate states each vertex generates. Subgraph enumeration performs a cartesian product between the current set of intermediate states and the adjacency lists to produce new states, thus generating an exponential amount of intermediate states. The enumeration of all induced subgraphs with five vertices ends when the cartesian product of L_4 finishes for each starting vertex.

One may consider GunRock to implement the subgraph enumeration depicted in Figure 3.3. We can initialize the frontier set using all vertices, the compute step can produce the intermediate states through cartesian products, the update phase can insert each intermediate state with less than four vertices into the frontiers to continue enumeration, and the convergence step can stop enumeration if no intermediate states are available. The problem is that, in this workflow, storing all intermediate states to update the frontier using updated values produced by the compute phase is mandatory. However, keeping all intermediate states for subgraph enumeration is unfeasible due to the combinatorial explosion in the intermediate states. Using GunRock's workflow to implement subgraph enumeration is equivalent to using the BFS traversal strategy, which inherently limits the enumeration of larger subgraphs in real-world datasets [54, 12]. GunRock implements the subgraph matching algorithm, which uses subgraph enumeration to return all subgraphs matching a pattern in a graph. However, our tests showed that this application works only for the triangle counting pattern. This seems to be an unresolved issue of GunRock's

Figure 3.3: Graph pattern mining execution workflow.



Source: created by the author.

API[1].

In summary, the state-of-the-art subgraph enumeration systems for GPU do not fully exploit GPU's computing power and can not execute subgraph enumeration when a diverse set of patterns with more than four vertices is used as input. Next, we present DuMato, our GPU-friendly subgraph enumeration system, which will be used to design and implement our novel strategies for efficient subgraph enumeration on GPU.

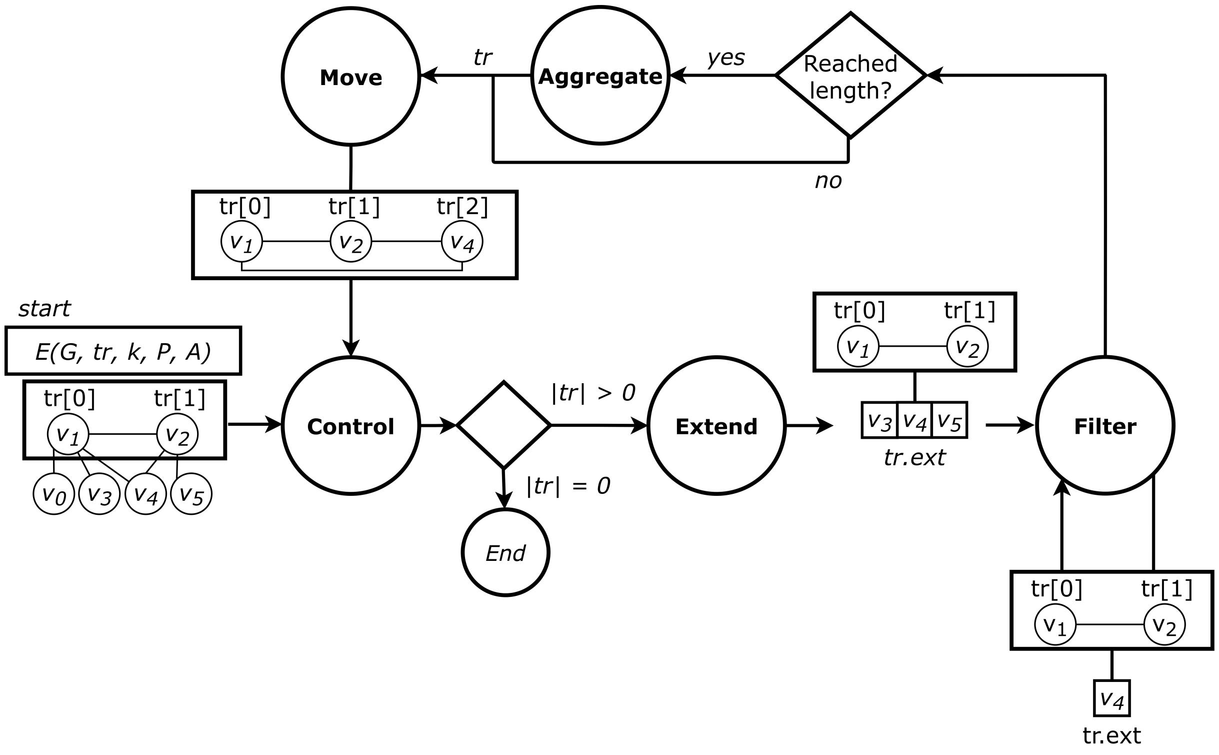
[1]https://github.com/gunrock/gunrock/issues/804

Chapter 4

DuMato Subgraph Enumeration System for GPUs

In this chapter we present *DuMato*, our subgraph enumeration system that supports a high-level implementation of GPM algorithms on GPU and will be used as a software platform to implement our novel strategies for subgraph enumeration on those platforms. The execution workflow of DuMato (Figure 4.1) employs the *filter-process* model [54], which allows the implementation of GPM algorithms based on an enumeration function E (Definition 13). Each circle refers to a phase in the enumeration workflow.

Figure 4.1: DuMato execution workflow.



Source: created by the author.

The process starts with a call $E(G, tr, k, P, A)$ 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 (step 3 in Def. 13). 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 (step 1 of Def. 13). These possible extensions are in the neighborhood of the current subgraph and are obtained from the adjacency of vertices in the traversal (Def. 8). 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 property P. 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. This phase is also responsible for compacting the extensions array such that invalid values are removed, and the extensions are reorganized in a contiguous memory/array. The Filter phase outputs the current traversal and extensions that are valid.

After all filters are executed, if traversal size reaches the target number of vertices, they are forwarded to the enumeration output (step 2 of Def. 13). This is accomplished in the *Aggregate* phase, in which traversals are consumed for counting, pattern counting, or buffering. The Aggregate phase is skipped if the traversal has not reached the target number of vertices.

Further, the *Move* phase decides whether to move forward or backward in the subgraph exploration. Moving forward means that an unprocessed extension is appended to the current traversal for processing (recursion call). Moving backward means that all extensions of the current traversal have been processed, and the algorithm can go back to 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 4.1.

DuMato's workflow (Fig. 4.1) can represent any GPM algorithm relying on the enumeration of induced subgraphs. Table 4.1 shows DuMato's API specification that may be used to create GPM algorithms. The functions receive as a parameter a data structure holding runtime information about the active traversal and extension arrays (TE, detailed later in Fig. 5.3) along with additional parameters.

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	
`[A1]` *aggregate_counter(TE)*	Aggregate	
`[A2]` *aggregate_pattern(TE)*		
`[A3]` *aggregate_store(TE)*		

Table 4.1: DuMato API.

Control and *Move* phases keep the workflow active while unprocessed traversals are in the search space. Because this loop-based exploration is familiar to most GPM algorithms searching for multiple subgraphs, 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.

The *Extend*, *Filter*, and *Aggregate* phases enable a straightforward and efficient representation of application-specific semantics on GPUs. The 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 may be used to generate extensions using alternative strategies that may be more effective in exploring subgraphs having patterns known apriori [15, 28]. `[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), applied to each extension to maintain only the valid ones. This interface can be used to design custom subgraph filters of extensions based on canonical candidate generation [54], density [37], and subgraph matching [22], among others. `[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 may be used, for instance, for subgraph counting [15] and scoring [25].

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 in preparation for the next iteration. This is common to most GPM algorithms and can be observed in lines 10 and 19 of Algorithm 9, 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. Consequently, new algorithms with new extend, filtering and aggregation demands may be implemented by replacing those lines.

Next, we detail how the enumeration phases (Figure 4.1) and the API (Table 4.1) of DuMato may be used to design and implement the *clique counting* and *motif counting* algorithms.

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 to every vertex in the traversal, the *Extend* phase generates the array of extensions from the neighbors of a single vertex in the traversal. `[EX]` call in line 3 of Algorithm 9 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

```
1    void clique_counting(TE){
2     while(control(TE)){
3 ●    if(!extend(TE, 0, 1)){
4 ●     u ← TE[TE.len − 1].id;
5 ●     filter(TE, &lower_than , [u]);
6 ●     filter(TE, &is_clique, []);
7      }
8      if(TE.len == k − 1)
9 ●     aggregate_counter(TE);
10     move(TE, false);
11    }
12   }

13   void motif_counting(TE){
14    while(control(TE)){
15 ●   if(!extend(TE, 0, TE.len))
16 ●    filter(TE, &is_canonical, []);
17     if(TE.len == k − 1)
18 ●    aggregate_pattern(TE);
19     move(TE, true);
20    }
21   }
```

Algorithm 9: Clique and Motif Counting algorithms.

$[0, 1)$). Given this set of extensions, `[FL]` is used in line 5 to invalidate non-canonical candidates (extensions lower than the last vertex), 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_than* 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, traversals with k vertices may be aggregated with `[A1]`, accumulating the length of the array of extensions in a counter.

Motif counting. A motif of size k is a pattern containing k vertices. The motif counting problem seeks to count the number of all possible motifs 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 15 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 16 the algorithm calls `[FL]` to invalidate extensions that, combined with the traversal, do not generate canonical candidates (Def. 12). The custom function *is_canonical* can be implemented using standard canonical filtering algorithms [54]. Finally, one `[A2]` call extracts the pattern from traversals combined with last-level extensions to increment the respective pattern-specific counters (line 19).

In summary, DuMato provides an execution workflow that may be used to design and implement GPM algorithms using subgraph enumeration. In the next chapter, we use DuMato's workflow to present our novel strategies for efficient subgraph enumeration on GPUs, thus mitigating the challenges associated with using this architecture as the target platform for GPM algorithms.

Chapter 5

Strategies for Efficient Subgraph Enumeration on GPUs

In this chapter we present our novel strategies for efficient subgraph enumeration on GPUs using DuMato as software platform. Table 5.1 maps where each strategy is demanded in DuMato's execution workflow.

`DuMato Enumeration Phase`	`Strategy`		
	`DFS-wide`	`Warp-centric Modeling`	`Load-balancing`
Extend	✓	✓	
Filter		✓	
Aggregate		✓	
Move	✓	✓	
Control		✓	✓

Table 5.1: Mapping of our novel strategies to DuMato's execution workflow.

The warp-centric modeling of execution steps (Section 5.2) is used in all enumeration phases, as it aims to provide more regular execution to mitigate the challenge of irregularity throughout the workflow, improving the efficiency of GPU's SIMT execution model. The DFS-wide subgraph exploration strategy (Section 5.1) mitigates the challenge of memory demand caused by the combinatorial explosion, and is used in the *Extend* phase to generate the intermediate enumeration states and in the *Move* phase to move forward or backward in the enumeration. The load-balancing layer (Section 5.3) mitigates the load imbalance caused by irregular parallel processing, and is implemented by the *Control* phase of the workflow.

5.1 DFS-wide Subgraph Exploration

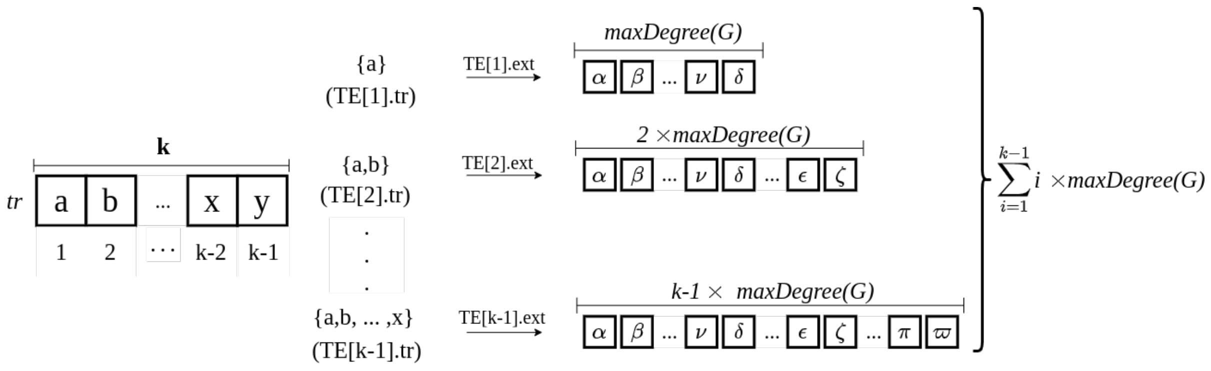
Algorithm 10 depicts the pseudocode of subgraph enumeration using BFS, DFS, and our novel DFS-wide traversal strategy. For these enumeration algorithms, assume we visit all subgraphs with k vertices of the input graph G, and the initial traversal tr is a single vertex of G. While BFS materializes all intermediate states throughout enumeration (line 9) and DFS enumerates recursively accessing only the current enumeration state (line 16), DFS-wide uses a data structure to store the intermediate states called TE (*Traversal Enumeration*), depicted by Figure 5.1. $TE[i].tr$ stores the $i-th$ vertex id of the traversal, $TE[i].ext$ stores the extensions generated by the traversal $\{TE[0].tr \cdots TE[i].tr\}$, and $TE[i].eg$ (*extensions generated*) is a flag that indicates whether the extensions of $\{TE[0].tr \cdots TE[i].tr\}$ have already been generated. The size of TE is predictable and proportional to the maximum degree of the graph.

```
 1 Function Enumerate_BFS(tr, k, G):
 2     states ← tr
 3     while states ≠ ∅ do
 4         cur ← pop(states)
 5         if |cur| == k then
 6             A(tr)
 7             continue
 8         next ← cur × N(cur)
 9         states.push(next)
10     end

11 Function Enumerate_DFS(tr, k, G):
12     if |tr| == k then
13         A(tr)
14         return
15     foreach v ∈ N(tr) do
16         DFS({tr + v}, k)
17     end

18 Function Enumerate_DFS_Wide(tr, k, G):
19     for i ∈ [1 ··· k − 1] do
20         TE[i].ext ← array(i × max(G))
21         TE[i].eg ← false
22     end
23     TE[1].tr ← tr
24     i ← 1
25     while i ≠ 0 do
26         if i == k then
27             A(tr)
28             i − −
29             continue
30         end
31         if !TE[i].eg then
32             TE[i].eg ← true
33             TE[i].ext ← N(tr[0 ··· i])
34         end
35         if TE[i].ext == ∅ then
36             i − −
37         else
38             TE[i + 1] ← pop(TE[i].ext)
39             i + +
40         end
41     end
```

Algorithm 10: DFS, BFS and DFS-wide traversal strategies.

Line 20 initializes the extensions and the flags to indicate the extensions that have not been generated yet. Starting from the first vertex (line 23), as in BFS, DFS-wide generates the set of all possible extensions using the neighbourhood of the traversal (line 33). However, different from BFS, DFS-wide does not visit all these extensions at once, and chooses only one extension from the current set of extensions to move forward and continue the enumeration (line 38). This approach not only presents a limited memory consumption but also allows memory parallelism and locality by accessing and storing an entire set of adjacency lists.

Figure 5.1: DFS-wide's TE data structure.



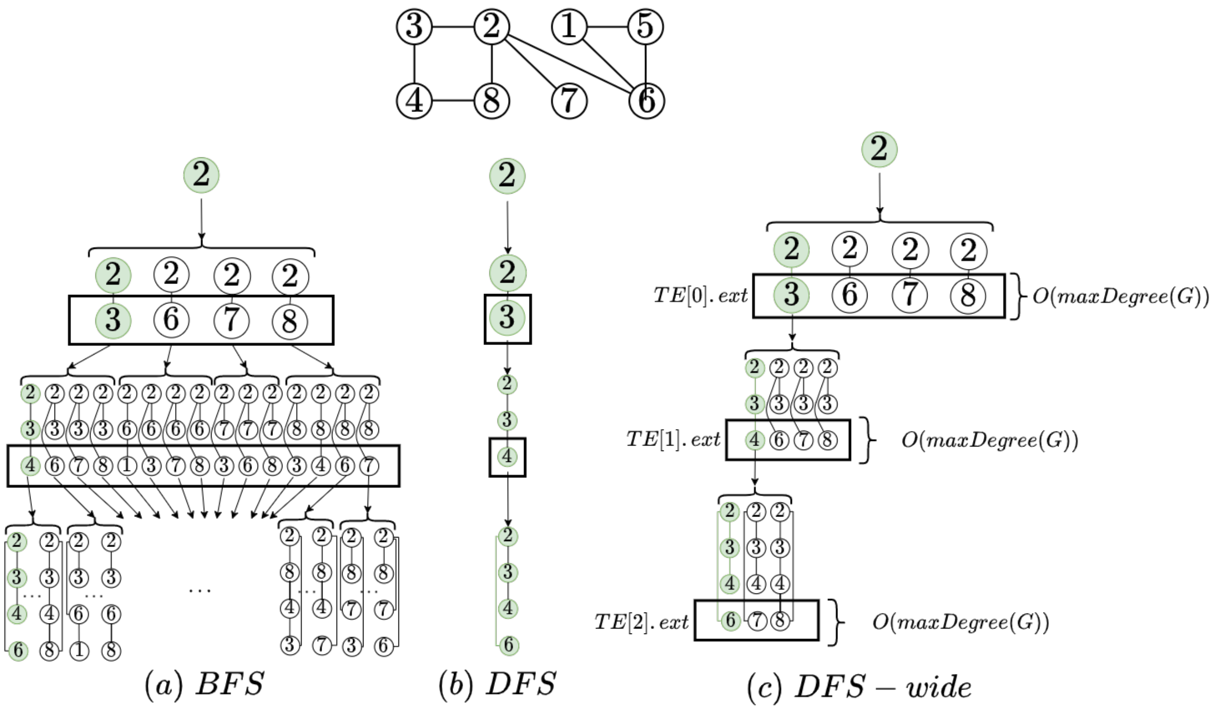
Source: created by the author.

Figure 5.2 depicts the enumeration lattice generated to visit the subgraph $\{2, 3, 4, 6\}$ using BFS, DFS, and DFS-wide subgraph exploration strategies. In BFS, all intermediate traversals throughout enumeration are materialized prior to visiting $\{2, 3, 4, 6\}$. Although this materialization may be implemented through regular memory accesses, the combinatorial explosion makes BFS demand too much memory and impairs its use in the enumeration of bigger subgraphs. On the other hand, DFS generates the minimum amount of intermediate traversals prior to visit $\{2, 3, 4, 6\}$. Despite its low memory consumption, the memory access pattern of DFS throughout enumeration is more sparse and deteriorates memory locality.

DFS-wide provides a good tradeoff between regularity and memory demand. As BFS, DFS-wide visits the entire neighborhood of the current traversal and stores the set of valid intermediate traversals. These intermediate traversals can be generated using coalesced memory requests, and the memory consumption is limited by the maximum degree of the input graph. However, as in a DFS exploration, DFS-wide moves forward in the enumeration by choosing only one possible traversal. The amount of materialized intermediate states throughout enumeration does not grow exponentially as in BFS.

Figure 5.3(a) presents an overview of the DFS-wide exploration steps, and Figure 5.3(b) shows the operations performed in one iteration of the BFS and DFS phases. In Figure 5.3(a), 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

Figure 5.2: BFS, DFS, and DFS-wide.

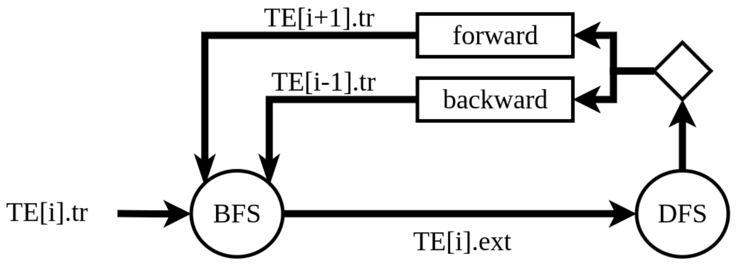


Source: created by the author.

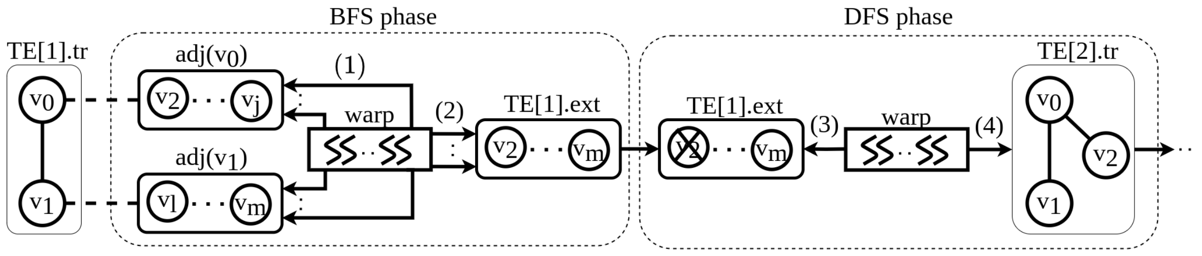
will be cached. The DFS receives the extensions and decides to move forward or backward in the enumeration, depending on the length of *tr* and the extensions. Note that, in both forward and backward enumeration, the DFS phase will access extensions in a contiguous memory that is probably cached, improving 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 5.3(b) 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 the current traversal (step 1), copies to extensions, and keeps only the unique extensions that are neither in the traversal nor in the extensions (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 5.2.2) of DuMato workflow, and the DFS phase is implemented by the *Move* phase, depicted by Figure 5.4. *Move* phase allows the warp to move forward/backward in the enumeration of a traversal, and its pseudocode is depicted in Algorithm 11. It receives *TE* and a flag *genedges* to indicate whether the edges of traversals should be generated during enumeration. The edges of traversals are useful in algorithms such as motif counting, and *Move* phase discovers them gradually as enumeration moves forward. If the current traversal still has not reached the size limit and the current set of extensions

Figure 5.3: DFS-wide subgraph exploration.



(a) Overview.



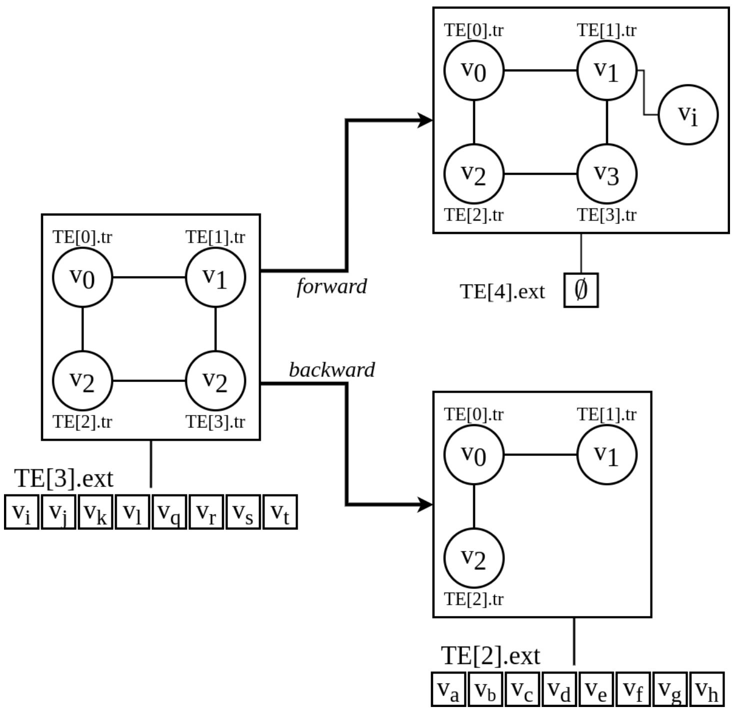
(b) BFS and DFS steps.

Source: created by the author.

is not empty (line 3), the warp moves forward in the enumeration by consuming an extension and extending the current traversal (lines 4-7). If the edges of traversal are needed, *induce* function (line 8) is an SIMD step that reuses the edges of the current traversal to produce the edges of the extended traversal. If the current traversal has either 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 (lines 12-13). In case the enumeration of the current traversal finishes (line 14), the warp pulls a new traversal (line 15). As all threads within a warp manipulate the same traversal and the primary purpose of Move is to update information about current traversal, it is an SISD phase, and only *induce* function that is costly is an SIMD step.

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 and 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 the current traversal. Although DFS consumes less memory than DFS-wide, it hinders parallelism, memory coalescence, and regular execution throughout subgraph enumeration.

Figure 5.4: *Move* phase.



Source: created by the author.

```
void move(TE, genedges){
  extensions ← TE[TE.len − 1].ext
  if(TE.len ≠ k − 1 && extensions ≠ ∅){
    extension ← extensions[extensions.len − 1];
    extensions.len − −;
    TE[TE.len].tr ← extension;
    TE[TE.len].eg ← false;
    TE.len + +;
    if(genedges)
SIMD    induce(TE);
  }
  else
    TE.len − −;
  if(TE.len == 0)
    TE ← pull_traversal(T);
}
```

Algorithm 11: Move primitive

The following section explains our efficient *warp-centric* enumeration phases, which uses the DFS-wide traversal structure to perform subgraph enumeration on GPUs through a more regular execution and memory access pattern.

5.2 Warp-centric Enumeration Phases

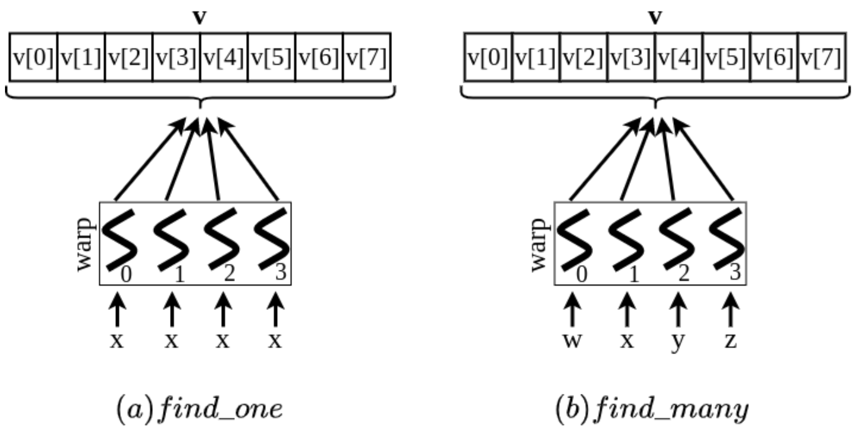
This section describes the design and implementation of the DuMato enumeration phases using the warp-centric programming model [24]. This model can be used in irregular algorithms to improve the regularity of the execution. In our design, a warp receives an initial traversal *tr* and all threads within the same warp perform subgraph enumeration starting from *tr* cooperatively. Threads within a warp alternate between SIMD and SISD phases throughout the execution workflow, allowing a more regular execution and memory access pattern. 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 by the DFS-wide strategy.

The following section presents the three warp-centric core primitives used to design and implement the enumeration phases: *find_one*, *find_many*, and *write*.

5.2.1 Warp-centric Core Primitives

For the sake of simplicity and without loss of generality, in this section we assume GPU warps with four threads. Figure 5.5 depicts the primitives warps use to find values efficiently in an array. Warps can use both primitives to search for values in the adjacency lists and extensions. Primitive *find_one* is used when threads within a warp need to find whether the given value x is present in an array v, and Algorithm 12 depicts its pseudocode. The variable *fnd_local* stores whether the x value was found in v by the current thread, and the variable *fnd_global* stores whether any thread within a warp found x in v. The main loop (line 3) iterates through v in parallel, and each thread within a warp receives a different value of v to compare with x (line 4). Primitive *any_sync* (line 5) is a CUDA warp exchange primitive and threads within a warp exchange their variable *fnd_local*. In case any *fnd_local* variable is not 0, *any_sync* returns 1 for all threads within a warp and sets 1 to *fnd_global* for all threads. Otherwise, all threads receive 0 in *fnd_global* and continue searching until any thread finds x or all elements in v are visited.

Figure 5.5: Find primitives.


v
v[0] v[1] v[2] v[3] v[4] v[5] v[6] v[7]
warp
0 1 2 3
x x x x
(a) find_one
v
v[0] v[1] v[2] v[3] v[4] v[5] v[6] v[7]
warp
0 1 2 3
w x y z
(b) find_many

Source: created by the author.

```
     $int\ find\_one(x,\ v,\ start,\ end)$:
SISD $ws \leftarrow warp\_size$;
SIMD $fnd\_local \leftarrow fnd\_global \leftarrow 0$;
SIMD $for\ (pos \leftarrow start + lane\,;\ pos < end\ \&\&\ !fnd\_global\ ;\ pos\ {+}{=}\ ws)$:
SIMD   $fnd\_local \leftarrow v[pos] == x$;
SIMD   $fnd\_global \leftarrow any\_sync(fnd\_local)$;
SISD **return** $fnd\_global$;
```

Algorithm 12: Primitive *find_ one.*

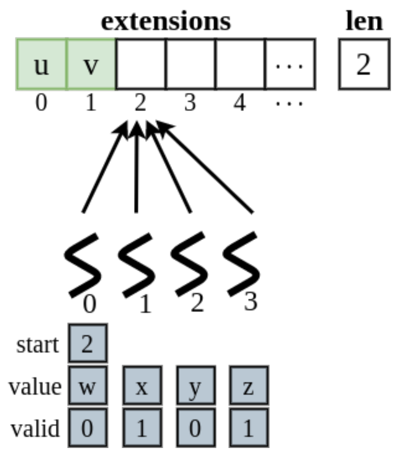
The primitive *find_ many* is used when threads within a warp need to find different values in an array v, and Algorithm 13 depicts a pseudocode of it. The algorithm is similar to *find_ one*, but with two crucial differences: variable *fnd_ global* stores a mask such that the i-th bit stores whether 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.

```
     $int\ find\_many(value,\ v,\ start,\ end)$:
SIMD $fnd\_local \leftarrow fnd\_global \leftarrow 0$;
SIMD $for\ (pos \leftarrow start\,;\ pos < end\ \&\&\ fnd\_global\ != \mathrm{0xffffffff}\,;\ start{+}{+})$:
SIMD   $found\_current \leftarrow v[pos] == value$;
SIMD   $fnd\_local \leftarrow fnd\_local\ ||\ found\_current$;
SIMD   $fnd\_global \leftarrow ballot\_sync(fnd\_local)$;
SIMD **return** $fnd\_local$;
```

Algorithm 13: Primitive *find_ many.*

The last core primitive is the *write* primitive, depicted in Figure 5.6. This primitive is used when threads within a warp have different values to be written in the extensions, but some may be invalid due to previous filtering. Algorithm 14 depicts a pseudocode of the *write* primitive. The function receives the set of extensions, the starting position where extensions should be written, the value itself, and a boolean indicating whether the extension 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 5) and invalid (line 6) values that the threads with lower lane will write, and threads use this information to calculate the exact position the valid values will be written in the extensions (lines 6 and 7).

Figure 5.6: Primitive *write*.



Source: created by the author.

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

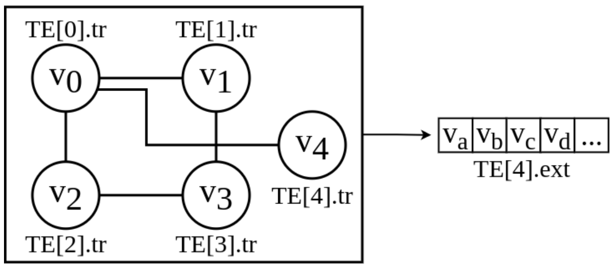
Algorithm 14: Primitive *write*.

The following section presents the warp-centric implementation of each subgraph enumeration phase using these three core primitives.

5.2.2 Extend

Figure 5.7 depicts the goal of the *extend* phase. This phase is the BFS step that generates the neighborhood extensions of a traversal *tr* by visiting the adjacency lists of a specific range of vertices. This range is essential 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).

Figure 5.7: *Extend* phase.



Source: created by the author.

Algorithm 15 shows our warp-centric implementation of the *extend* phase. Lines 5-6 are an initial SISD phase, where all threads within the warp receive a vertex id prior to visiting its adjacency list. Lines 9-21 correspond to the SIMD phase in which threads within the warp visit an adjacency list in parallel (line 10), and vertices already in either current traversal or extensions are considered invalid (lines 11-14). At last, each thread within the warp writes its extension to the extensions set in parallel (line 15). 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 helpful to avoid unnecessary calls to the *filter* phase. 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.

5.2.3 Filter

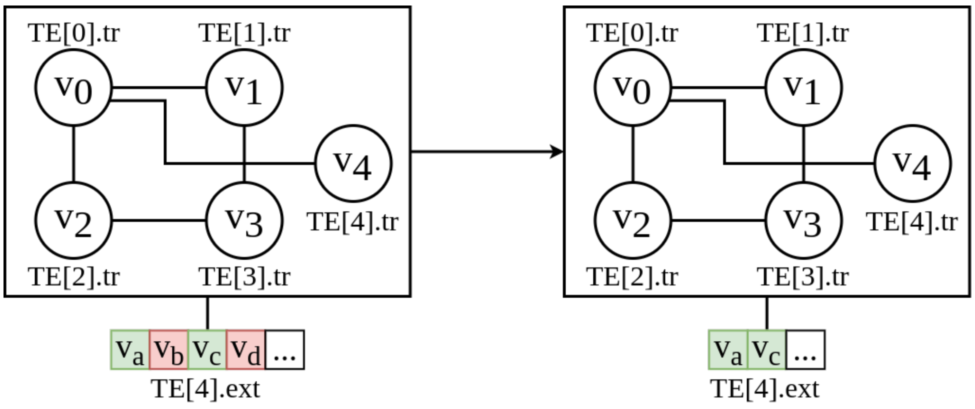
Figure 5.8 depicts the goal of the *filter* phase. Given a traversal tr, *filter* phase iterates the current extensions of tr in parallel and removes those that do not satisfy a property (P function, Def. 13). Algorithm 16 shows our warp-centric implementation of the *filter* phase. The algorithm gets the current traversal and a function pointer P, which indicates whether an extension is valid. Each thread within the warp gets an extension (line 5) and passes it to the P function (line 6), which may lead to its invalidation depending on the implementation of P. For example, one of the filters used in the clique counting algorithm checks whether the id of an extension is lower than the id of tr's last vertex, and it is implemented by a call to $filter$ phase along with a function pointer P that performs this test. P functions are warp-centric and can be implemented using DuMato primitives to access the TE data structure. Lines 7-8 write the extensions back

```
1         $boolean\ extend(TE, start, length)\{$
2  SISD   $eg \leftarrow TE[TE.len-1].eg;$
3  SISD   $if(!eg)\{$
4  SISD     $TE[TE.len-1].eg \leftarrow true;$
5  SISD     $for(i \leftarrow start\ ;\ i < length\ ;\ i++)\{$
6  SISD       $id \leftarrow TE[i].tr;$
7  SISD       $d \leftarrow degree(id);$
8  SISD       $produced \leftarrow 0;$
9  SIMD       $for(j \leftarrow lane\ ;\ j < d\ ;\ j+= warp\_size)\{$
10 SIMD         $ext \leftarrow adj(id,\ j);$
11 SIMD         $inTr \leftarrow find\_many(ext, TE[\,].tr, 0, TE.len-1);$
12 SIMD         $cont \leftarrow any\_sync(!inTr);$
13 SIMD         $inExt \leftarrow cont\,?\,find\_many(ext, TE[\,].tr, 0, TE.len-1)\ :\ false;$
14 SIMD         $valid \leftarrow !inTr\ \&\&\ !inExt;$
15 SIMD         $write(TE[TE.len-1].ext,\ produced,\ valid,\ ext);$
16            $\}$
17 SISD       $produced+= d;$
18          $\}$
19        $\}$
20 SISD   $return\ eg;$
21        $\}$
```

Algorithm 15: *Extend* primitive.

to the extensions array, keeping only the valid ones. Lines 9-10 control the actual amount of valid extensions after filtering.

Figure 5.8: *Filter* primitive



Source: created by the author.

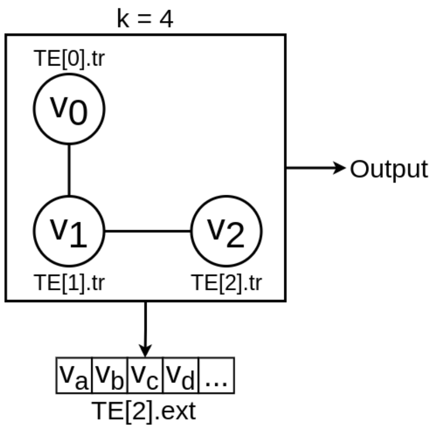
```
     $void\ filter(TE,\ P,\ args)\{$
SISD   $e \leftarrow TE[TE.len - 1].ext;$
SISD   $produced \leftarrow 0;$
SIMD   $for(i \leftarrow lane;\ i < extensions.len\ ;\ i+ = warp\_size)\{$
SIMD     $extension \leftarrow extensions[i];$
SIMD     $valid \leftarrow P(TE, extension, args);$
SIMD     $TE[TE.len - 1].ext.len\ - =\ warp\_size;$
SIMD     $write(TE[TE.len - 1].ext,\ produced,\ valid,\ extension);$
SIMD     $amount \leftarrow popc(ballot\_sync(valid));$
SIMD     $produced \leftarrow produced + amount;$
       $\}$
     $\}$
```

Algorithm 16: Filter primitive.

5.2.4 Aggregate

Figure 5.9 depicts the goal of the *aggregate* phase. 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 Definition 13). *DuMato* provides three aggregation primitives: *aggregate_pattern*, *aggregate_counter* and *aggregate_store*, as defined in Table 4.1, which are explained next.

Figure 5.9: *Aggregate* phase.



Source: created by the author.

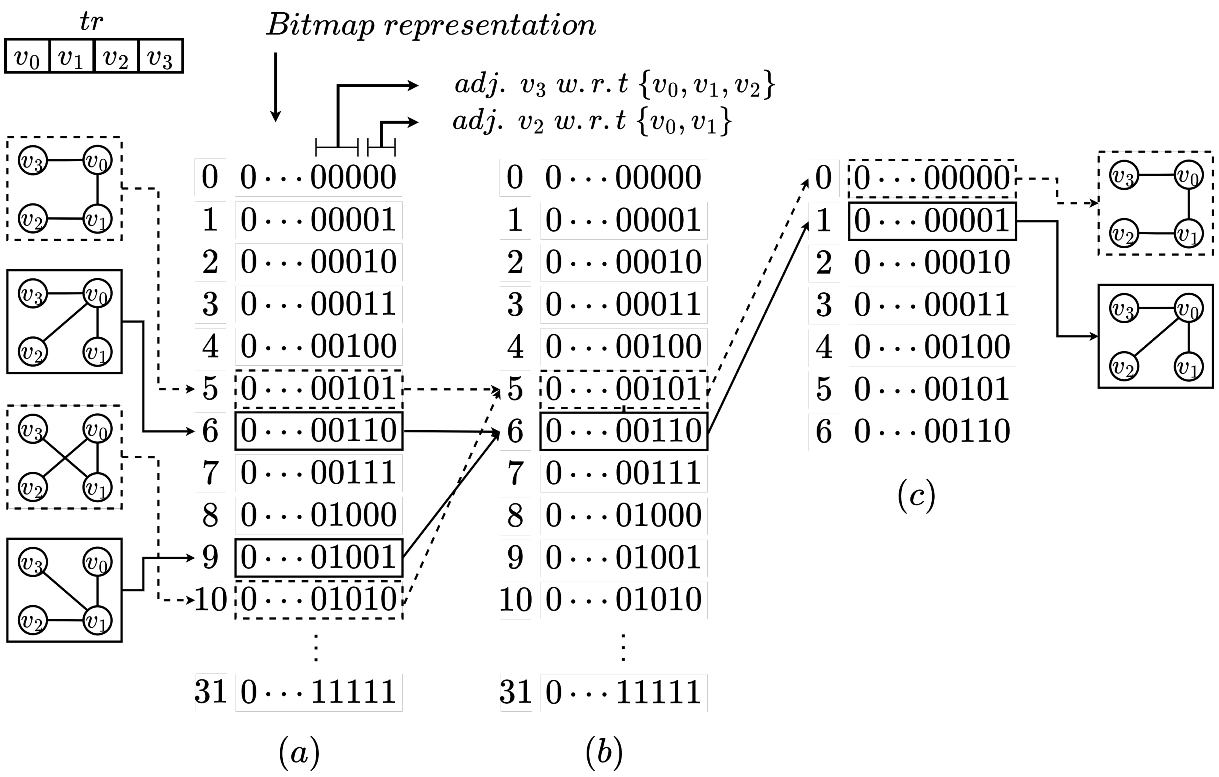
The *aggregate_pattern* is the most challenging primitive for implementation on GPUs. It is used when the output of the GPM algorithm relies on counting the occurrence of patterns with k vertices, such as motif counting. This is executed on a per-warp basis, such that each warp performs *canonical relabeling*, converting each subgraph with k vertices to its canonical representative and incrementing a counter. This is only possible due to our novel representation of patterns, which reduces the memory required to store them. The solution for canonical relabeling relies on graph isomorphism, and GPM systems (including Pangolin [12]) perform it on CPU using tools such as Nauty [41]. To the best of our knowledge, we are the first work to implement canonical relabeling on GPU.

Figure 5.10 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 $\{v_0, v_1, v_2\}$ (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 Figure 5.10(a). Each possible traversal with 4 vertices can be mapped to its canonical representative, shown in Figure 5.10(b). As traversals often produce isomorphic subgraphs, different traversals may be mapped to the same canonical representative. The amount of patterns is much smaller than the number of possible traversals, as seen in Figure 5.10(c), and the bitmap representation of patterns may 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 5.10). This is performed 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 patterns 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 [22]). DuMato provides this dictionary as an input file.

The *aggregate_counter* primitive is called when the desired results/output of the GPM algorithm is pattern counting, such as in the clique counting algorithm. Each warp produces its 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 the warps' counters afterward on the CPU. This is a simple and inexpensive computing 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 match 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.

Figure 5.10: Canonical relabeling on GPU.



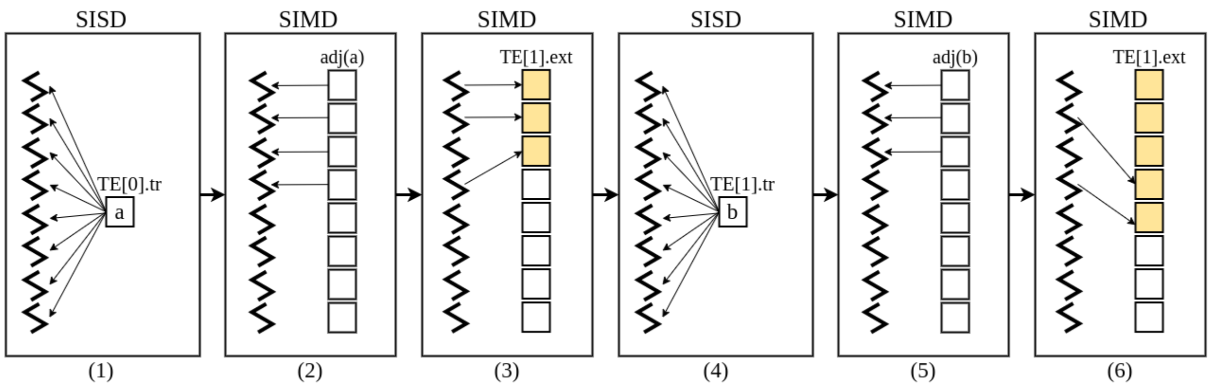
Source: created by the author.

5.2.5 Warp Virtualization

Divergent executions are one of the impacts of irregularity. In the warp-centric programming model, all threads within a warp work cooperatively to make progress in a task. Thus, keeping all threads within a warp working in the task is crucial to reduce divergences, achieve massive parallelism and improve the task throughput. Figure 5.11 uses a traversal $TE.tr = \{a, b\}$ to depict the activity of threads within a warp in our warp-centric design of the *extend* phase. For this example, assume warps with eight threads and vertices a and b with degree four. Step 1 is an SISD step, and all threads in the warp receive the id of the vertex to be visited. Step 2 visits the adjacency of a, and step 3 writes in the extensions the vertices that belong to the neighborhood of $TE.tr$. In step 2, half of the warp's threads are idle because a has four neighbors, and the warp has eight threads. Once the adjacency of a is visited and vertices in the traversal's neighborhood are written in the extensions, the steps are repeated using vertex b. Step 4 informs the warp that vertex b will be visited, step 5 visits the adjacency of b, and step 6 writes in the extensions. Again, half of the warp's threads are idle in step 5.

SIMD steps 3 and 5, depicted in Figure 5.11, degrade the warp efficiency of our

Figure 5.11: Warp-centric steps of the *extend* phase.

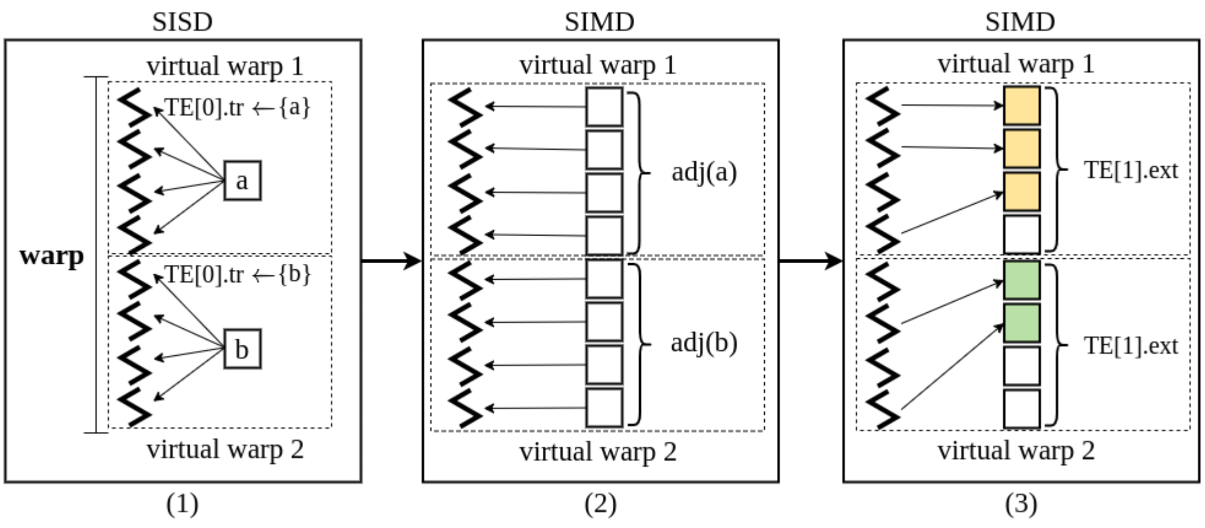


Source: created by the author.

execution, as part of the threads in the warp is idle. This happens because, in practice, the size of the adjacency lists is not multiple of the hardware warp size (32). Although the actual warp size can not be changed, we can create a virtual warp size and split the warp into virtual independent execution units [24]. Recent advances in GPU hardware scheduling also allow independent thread scheduling [43], and each thread in a warp may have its program counter. Thus, virtual warps can run in parallel.

We propose an extension of our warp-centric enumeration steps to allow warp virtualization, as depicted by Figure 5.12. We set the size of the virtual warps (prior to the execution) and traversals are assigned to virtual warps rather than physical ones. Note that, using virtual warp sizes, we reduce the chances of idleness of threads within the same warp, thus providing oportunities to increase GPU's massive parallelism.

Figure 5.12: Warp-centric steps of the *extend* phase using virtualization.



Source: created by the author.

In order to implement warp virtualization inside DuMato, we have to change the synchronization granularity between threads within a warp. In the standard warp-centric version, whenever any synchronization was necessary, we knew all threads within a warp were supposed to wait together. For example, in line 5 of the *find_one* primitive depicted in Algorithm 12, the primitive *any_sync* checked the value of the *fnd_local* variable of all threads within the same physical warp (32 threads). When we have virtual warps, one thread can not wait for information from all other threads within the same physical warp, as they may belong to different virtual warps. Thus, each thread we must be aware of which other threads within the same physical warp belong to the same virtual warp.

In order to do that, we create a 32-bit per-thread synchronization mask to indicate which threads are working together within a warp. For example, suppose virtual warps with 16 threads. Threads with lane in the range $[0 \cdots 15]$ would receive the mask $0x0000ffff$, while threads with lane in the range $[16 \cdots 31]$ would receive the mask $0xffff0000$. This mask is passed as a parameter for all DuMato functions, along with the size of the virtual warp (used in the for loops in the lockstep execution).

For example, Algorithm 17 depicts the primitive *find_one* using warp virtualization. The mask associated with each thread is passed along with the current virtual warp size as a parameter. In the main for loop (line 3), the lockstep execution is incremented according to the size of the virtual warp. Besides, the starting point of each thread is calculated using a virtual lane (a number in the range $[0 \cdots 15]$), which is a number that indicates the internal id of a thread within a virtual warp rather than the physical one ($pos = start + virtual\_lane$, line 3). Different from the standard warp-centric implementation, the *any_sync* primitive receives the mask as an extra parameter (line 5) to indicate which threads within the same physical warps are supposed to wait together for the synchronization step. We perform similar adequations in all DuMato primitives to implement warp virtualization.

```
1          int find_one_virtual(x, v, start, end, mask, warp_size):
2  SIMD   ws ← warp_size;
3  SIMD   vlane ← virtual_lane;
4  SIMD   fnd_local ← fnd_global ← 0;
5  SIMD   for (pos ← start + vlane ; pos < end && !fnd_global ; pos += ws):
6  SIMD     fnd_local ← v[pos] == x;
7  SIMD     fnd_global ← any_sync(fnd_local, mask);
8  SIMD   return fnd_global;
```

Algorithm 17: Primitive *find_ one* with virtualization.

5.3 Warp-level Load Balancing

The cost of enumerating different traversals may vary, which leads to load imbalance among warps. We propose a workload redistribution scheme to mitigate this problem. Our strategy makes decisions based on the warp level activity information, and each thread warp has an associated flag to indicate whether it is active or idle.

In our load-balancing approach, the CPU constantly and asynchronously reads the activity information from the GPU to decide whether the load should be redistributed to improve GPU utilization. When load balancing is to be performed, the CPU informs the GPU by setting a flag accessed by warps in the Control phase. If this flag is set, the warps stop their execution in a consistent state once they get in the Control phase. When all warps stop, the execution control is returned to the CPU for work redistribution. We highlight that, although it is possible to implement complex load balancing mechanisms using buffers to store jobs generated by warps [9], our CPU-only strategy mitigates synchronization overheads from GPU, and more resources can be allocated for subgraph enumeration.

Our load-balancing mechanism is implemented through the functions *when_ rebalance* and *how_ rebalance* depicted in Algorithm 18. 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.

```
1  void when_rebalance(DM_info gpu){
2    flags $\leftarrow$ gpu.read_flags();
3    while(flags.active_warps $> 0$) :
4      if(flags.idle_warps $>$ thr) :
5        gpu.lb $\leftarrow$ true;
6        gpu.waitKernel();
7        how_rebalance(gpu);
8        gpu.lb $\leftarrow$ false;
9        gpu.runKernel();
10     flags $\leftarrow$ gpu.read_flags();

11 void how_rebalance(DM_info gpu){
12   idles $\leftarrow$ list(gpu.idles);
13   actives $\leftarrow$ heap(gpu.actives);
14   total_weight $\leftarrow$ sum_weight(actives);
15   avg_weight $\leftarrow$ total_weight/|warps|;
16   for(i $\leftarrow 0$ ; i $<$ donations ; i++) :
17     for each(idle $\in$ idles) :
18       donator $\leftarrow$ actives.pop_heap();
19       idle.jobs.push(extract(donator));
20       if(donator.weight $>$ avg_weight) :
21         actives.push_heap(donator);
```

Algorithm 18: CPU code for load balancing.

The load-balancing mechanism consider the cost of the jobs (traversals) assigned to a warp and is able to donate several jobs among busy and idle warps. In order to do that, it selects several jobs from warp donators using information from their current traversal and extensions. The function *how_rebalance* depicts this strategy. We create a list of idle warps (line 12) and a max heap with the active ones (line 13). The criteria 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 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). In the Section 6.3.2 we perform an empirical evaluation to discover the appropriate amount of donations during the rebalancing.

In summary, this chapter presented our strategies to mitigate the main challenges for efficient subgraph enumeration on GPUs: the DFS-wide subgraph exploration reduces the impacts of combinatorial explosion while providing opportunities for regular memory requests; the warp-centric enumeration phases provide a workflow to minimize the effects of irregular processing on GPU SIMT execution model; our load-balancing layer proposes an efficient workload redistribution scheme to mitigate the load imbalance during parallel subgraph enumeration. In the next chapter, we evaluate the performance impacts of each strategy and compare our implementation of DuMato to the state-of-the-art subgraph enumeration systems.

Chapter 6

Experimental Evaluation

This section presents an extensive performance evaluation of each proposed optimization using DuMato subgraph enumeration system. We employ the implementations of *clique counting* and *motif counting* algorithms, as they represent two essential categories in GPM processing: the exploration of subgraphs sharing a single pattern (*clique counting*) and the exploration of subgraphs containing multiple patterns (*motif counting*).

All executions and analysis were performed using five different implementations of each algorithm. The characteristics of each implementation are summarized in Table 6.1. *DM_ DFS* (***DuM****ato* ***D****epth-****F****irst* ***S****earch*) uses the DuMato API to implement subgraph enumeration using standard DFS exploration. Each thread receives a different traversal and threads within a warp enumerate distinct traversals in parallel. None of our optimization strategies are implemented in this version, and it will be used as our GPU baseline to evaluate the performance impacts of our optimizations. *DM_ WC* (***DuM****ato* ***W****arp-****C****entric*) uses the DuMato API to implement subgraph enumeration and, different from *DM_ DFS*, each warp enumerates the same traversal in parallel cooperatively using our novel DFS-wide approach (Section 5.1) and the warp-centric design (Section 5.2). *HAND_ WC* (**Hand**crafted **W**arp-**C**entric) is the *DM_ WC* without using the DuMato API calls. This version is used to evaluate the overhead of DuMato API. *DM_ WCV* (***DuM****ato* ***W****arp-****C****entric with* ***V****irtualization*) is *DM_ WC* with our warp virtualization strategy enabled (Section 5.2.5). *DM_ WCLB* (***DuM****ato* ***W****arp-****C****entric with* ***L****oad-****B****alancing*) is the *DM_ WC* version with our warp-level load-balancing mechanism enabled (Section 5.3). *DM_ WCLB* version is our optimal implementation and will be used to compare DuMato with the state-of-the-art subgraph enumeration systems *Pangolin* [12], *Peregrine* [28] and *G2Miner* [11].

Version	DuMato Features			
	API	DFS-wide	Warp-centric	Load Balancing
DM_ DFS	✓			
DM_ WC	✓	✓	✓	
DM_ WCLB	✓	✓	✓	✓

Table 6.1: Characteristics of each implementation.

6.1 Experimental Setup

The characteristics of five real-world datasets used in our experiments are presented in Table 6.2. CPU experiments were conducted on a machine with an Intel Xeon Silver 4108 CPU (16 threads with hyperthreading), 48GB of RAM, and Ubuntu 18.04. GPU experiments used an NVIDIA TITAN V with 12GB and CUDA 10.1. The time limit adopted for each execution was 24 hours. Experiments that have not finished within 24 hours are marked with the signal "-". Every execution was run three times and demonstrated low variability (standard deviations in 0.06%-1.07%). The result is the average execution time of the three executions. We do not present the results for the LiveJournal graph for motif counting because the executions exceed our 24-hour limit even for small subgraph sizes ($k > 4$).

Dataset	**V(G)**	**E(G)**	**Avg. Degree**	**Density**	**Max. Degree**
Citeseer [18]	3264	4536	2.77	8.51×10^{-4}	99
ca-AstroPh [35]	18772	198110	21.10	1.12×10^{-3}	504
Mico [18]	96638	1080156	22.35	2.31×10^{-4}	1359
com-DBLP [59]	317080	1049866	6.62	2.08×10^{-5}	343
com-LiveJournal [59]	3997362	34681189	17.35	4.34×10^{-6}	14815

Table 6.2: Graphs used for evaluation.

6.2 Performance Improvements to Warp-Centric DFS-wide

Table 6.3 shows the execution times for *DM_ DFS*, *DM_ WC* and *HAND_ WC* as the size of the subgraphs mined (k) varies.

6.2.1 DuMato's Overhead

A comparison between *DM_ WC* and *HAND_ WC* shows the overhead of using DuMato to implement GPM algorithms. The handcrafted versions are a little faster than the versions implemented using DuMato, and this overhead is primarily caused by extra function calls when using an API-oriented application. The maximum overhead is about

6%, and there is a decrease in the overhead as we increase the value of k. We believe this overhead is negligible w.r.t. the overall execution time, mainly when we compare the comprehensiveness and readability of a GPM code written using DuMato to a handcrafted one.

6.2.2 Warp-centric vs. DFS

The *DM_ DFS* version assigns traversals per thread, and each thread enumerates their traversals independently. As processing each traversal may result in different execution paths, threads within a warp will diverge throughout 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 26× (Clique, Mico and $k = 5$) w.r.t. the *DM_ DFS*.

		System	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$
Motifs	Citeseer	`DM_DFS`	0.02	0.54	11.83	252.97	7.45K	-	-
		`DM_WC`	0.01	0.04	0.91	18.09	334.32	9.65K	-
	ca-Astr.	`DM_DFS`	1.79	590.63	-	-	-	-	-
		`DM_WC`	0.11	27.88	6.87K	-	-	-	-
	Mico	`DM_DFS`	23.74	15.74K	-	-	-	-	-
		`DM_WC`	1.24	768.10	-	-	-	-	-
	DBLP	`DM_DFS`	1.05	199.34	29.99K	-	-	-	-
		`DM_WC`	0.07	11.33	16.60K	-	-	-	-
Clique	Citeseer	`DM_DFS`	0.01	0.01	0.01	0.01	∅	∅	∅
		`DM_WC`	0.01	0.01	0.01	0.01	∅	∅	∅
	ca-Astr.	`DM_DFS`	0.21	4.35	48.0	329.32	2.82K	23.14K	-
		`DM_WC`	0.01	0.25	2.91	24.66	190.55	1.65K	12.27K
	Mico	`DM_DFS`	2.93	249.22	14.20K	-	-	-	-
		`DM_WC`	0.13	10.70	527.10	26.22K	-	-	-
	DBLP	`DM_DFS`	0.14	3.63	124.08	3.35K	74.60K	-	-
		`DM_WC`	0.01	0.21	6.74	196.36	4.78K	-	-
	LiveJr.	`DM_DFS`	291.0	5.30K	-	-	-	-	-
		`DM_WC`	14.01	219.65	6.17K	-	-	-	-

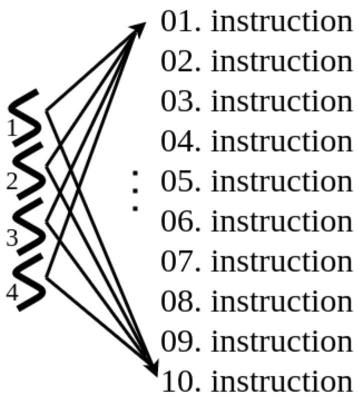
∅: no valid subgraphs

Table 6.3: Execution time (seconds) of `DM_DFS`, `DM_WC` and `HAND_WC`.

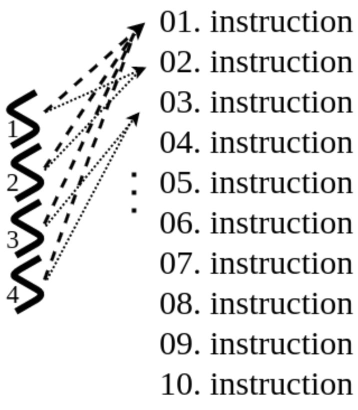
To understand the effects of our exploration and optimization strategies at the hardware level, Table 6.4 shows the improvements of *DM_ WC* over *DM_ DFS* using execution and memory metrics collected from CUDA NVProf profiling tool [45]. GPU profiling is much slower than standard runs, and we present the results using the DBLP dataset for k up to 4. Note that the DM_DFS and DM_WC versions perform the same computations but use different parallelization strategies and optimization techniques. Thus, they will handle the same processing demands during enumeration.

Metrics are divided into two categories: (i) Execution, which measures the efficient use of the 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. To better understand this metric, Figure 6.1 depicts the execution pattern of two different warps (assume warps with four threads). In Figure 6.1(a), the warp does not diverge and executes in lockstep during the execution of the entire set of instructions. In this case, the worst-case result of the *inst_per_warp* metric is ten. In Figure 6.1(b), the warp executes only the first instruction in lockstep. After this point, there is a divergence, and two blocks of threads within the same warp execute concurrently: threads $\{1, 2\}$ and threads $\{3, 4\}$. In this case, the worst-case result of the *inst_per_warp* metric is eighteen: 1 (first instruction) + 9 (threads {1,2}) + 8 (threads {3,4}). The more regular the execution is, the fewer divergent instructions are issued, and warps require fewer instructions.

Figure 6.1: Metric *inst_per_warp* and divergences.



(a) With divergence.



(b) Without divergence.

Source: created by the author.

For memory, we chose the metric *gld_transactions*, which measures the total amount of load transactions requested to global memory. The more coalesced the memory access pattern is, the fewer 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 the 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.

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 6.4: Improvements of `DM_WC` over `DM_DFS`.

This regularity is confirmed by the execution measures, 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 lock-step execution allowed threads to perform memory requests together using coalesced requests. Therefore, our *DM_ WC* version reduces the total number of memory transactions. This reduction is confirmed by the memory metric, with improvements ranging from 2.90x to 7.92x, showing that our memory optimizations reduce wasted bandwidth and improve memory efficiency.

6.2.3 Warp Virtualization

Table 6.5 shows the execution time of our warp-centric version using virtual warps with 16 and 8 threads, and compares it to the standard warp-centric version (warps with 32 threads) using different datasets. We show only the results for the motif counting application, as it accesses the entire set of adjacency lists during the *extend* phase of DuMato (see line 3 of Algorithm 9) and thus presents more chances of taking advantage of warp virtualization. Warp virtualization should increase warp efficiency by reducing the granularity of parallelism when visiting one adjacency list, thus reducing the chances of having idle threads within a warp. However, what we see is a performance deterioration as we increase the amount of virtual warps per warp. For all values of k, the standard warp-centric version is faster than the warp-centric with virtualization. The performance deterioration caused by warp virtualization is explained by two factors acting together: the way GPUs schedule virtual warps and the inherent load imbalance of subgraph enumeration.

App	Dataset	k	Warp size		
			32	16	8
Motifs	Citeseer	7	5,06	6,28	9,83
		8	96,94	111,39	182,81
	ca-Astroph	5	126,77	208,24	391,71
		6	23625,04	39339,01	74904,86
	Mico	4	23,27	38,34	70,42
		5	7621,53	13033,78	24978,10
	DBLP	5	31,90	48,32	86,43
		6	2648,65	4089,01	7500,62

Table 6.5: Comparison between warp-centric with and without warp virtualization.

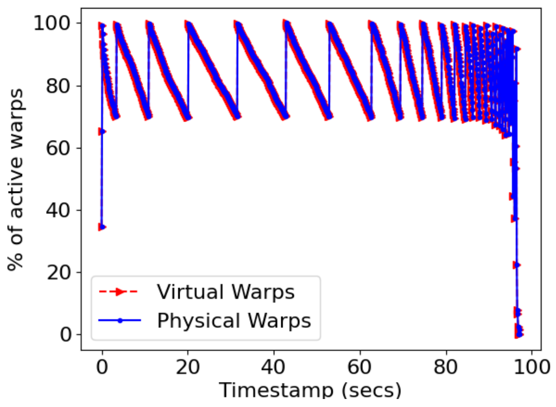
6.2.3.1 Scheduling of Virtual Warps

Virtual warps are independent computing units, which may reduce the chances of synchronization issues on GPU and facilitate the implementation of algorithms relying on fine-grained synchronizations. However, it is not guaranteed that virtual warps will be scheduled to execute concurrently. The GPU scheduler is conservative and assumes that independent virtual warps should not run concurrently [43]. Besides, as a physical warp may contain several virtual warps executing independently, there will be more misaligned memory access patterns within the same warp, and consequently an increase in the amount of memory transactions per warp, which reduces memory efficiency.

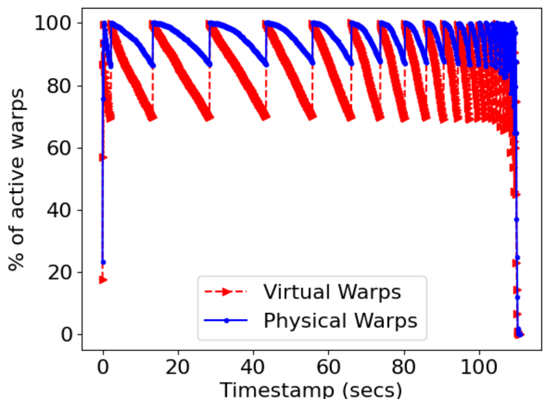
6.2.3.2 Impacts of Load Imbalance

Figure 6.2 shows the percentage of active physical warps and virtual ones for motif counting using 30% of the load-balancing threshold. In the standard warp-centric version (Figures 6.2(a) and 6.2(c)), a virtual warp represents a physical one, and the average percentage of active warps is kept above the threshold throughout the majority of the execution. In the warp-centric version using virtualization with 16 threads (Figures 6.2(b) and 6.2(d)), the percentage of active virtual warps is kept above the threshold throughout execution, as the load-balancing layer uses the amount virtual warps to decide when to rebalance. Two virtual warps may belong to different physical warps, and there is an increase in the amount of active physical warps throughout execution compared to the standard warp-centric version. Despite that, Figure 6.3 illustrates the average amount of active virtual warps per physical warp in two executions. Due to the inherent load imbalance of subgraph enumeration, the amount of physical warps with two active virtual warps decreases throughout execution. As the GPU scheduler does not guarantee that the amount of SPs allocated to a virtual warp is exactly the amount of its active threads, we do not have maximum SIMD efficiency when physical warps contain less than 32 active threads and virtual warps are spread throughout the grid. In other words: an increase in the average amount of physical warps depicted in Figures 6.2(b) and 6.2(d) does not reflect an actual better parallel performance, as many physical warps contain only half of their threads in active state.

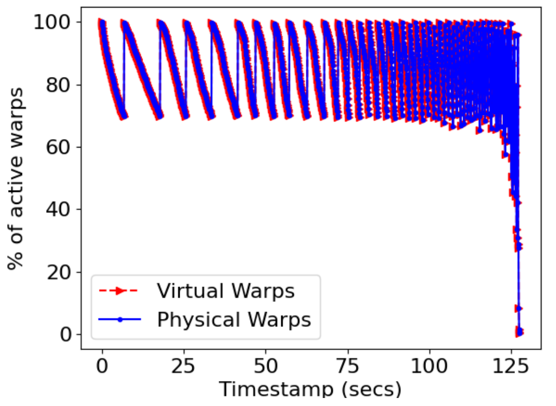
Figure 6.2: Warp activity of motif counting.



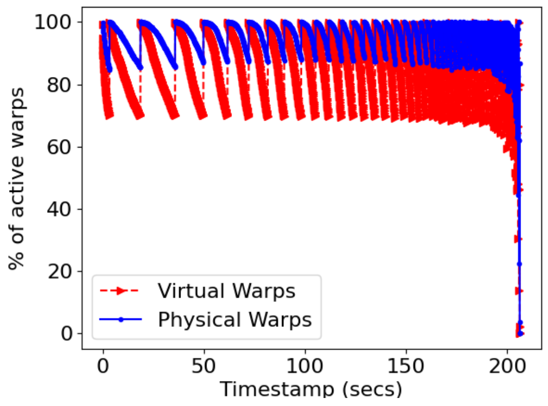
(a) Citeseer, $k = 8$, warp size 32



(b) Citeseer, $k = 8$, warp size 16



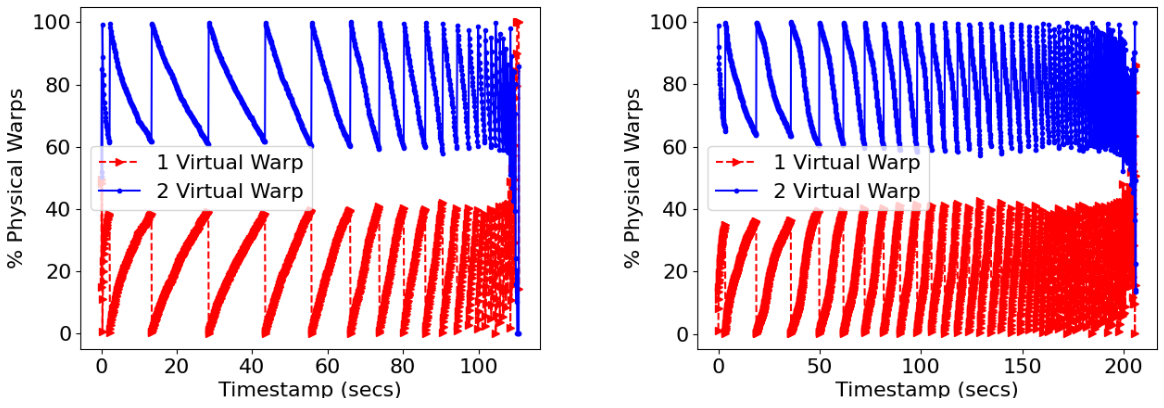
(c) ca-AstroPh, $k = 5$, warp size 32



(d) ca-AstroPh, $k = 5$, warp size 16

Source: created by the author.

Figure 6.3: Physical warp activity of motif counting.



(a) Citeseer, $k = 8$ (b) ca-AstroPh, $k = 5$

Source: created by the author.

6.3 Gains Due to Load-Balancing

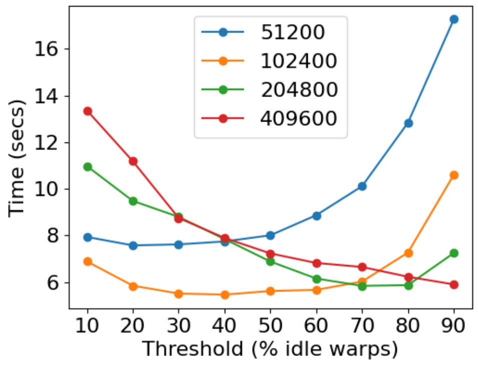
6.3.1 Number of Threads and Load-Balancing Threshold

Here, we analyze the effect of number of threads used and rebalancing threshold to the execution time for the motif counting application (results for clique counting are equivalent). Our evaluation varied the number of threads and threshold for all algorithms and datasets and several values of k, using the DM_WCLB version. The results are shown in Figures 6.4 and 6.5 for motifs and clique, respectively.

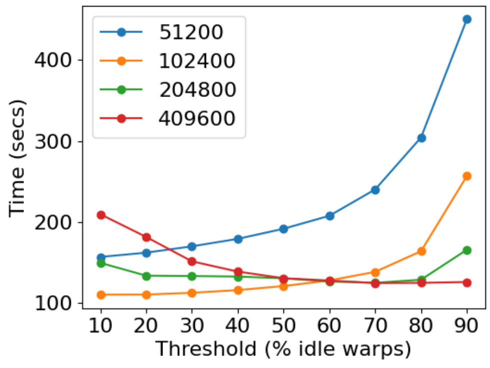
As can be observed, 51,200 threads are not sufficient to fully take advantage of the GPU, which attains better performance with higher thread count values. Specifically, the configuration with 102,400 threads and a threshold of 30% led to the best performance for most of our experiments and was, consequently, selected to be used in the rest of the experiments in this work. The configuration with 409,600 threads and a threshold between 80%-90% also provided good performance. However, the more threads we instantiate, the more registers are needed, and the chances of performance deterioration due to register pressure are higher especially when executed on a GPU with fewer physical registers. The only scenario where 102,400 threads and a threshold of 30% is far from the optimal execution performance is using ca-AstroPh and $k = 4$. The load becomes imbalanced more quickly for this small k, and a threshold of 30% using 102,400 threads results in several load-balancing calls that deteriorate performance.

In order to identify the tradeoff associated with the variation of the load-balancing

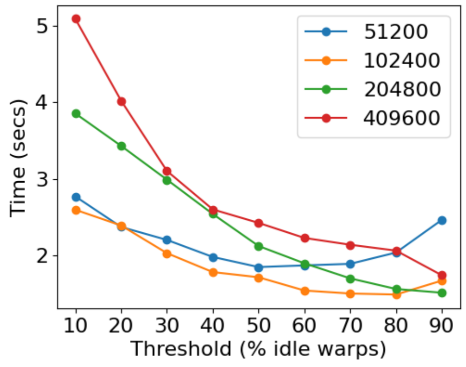
Figure 6.4: Impact of load-balancing threshold and number of threads to execution time (motif counting).



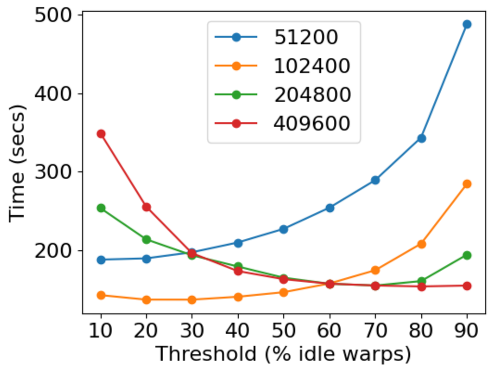
(a) Citeseer, $k = 7$



(b) Citeseer, $k = 8$



(c) ca-AstroPh, $k = 4$



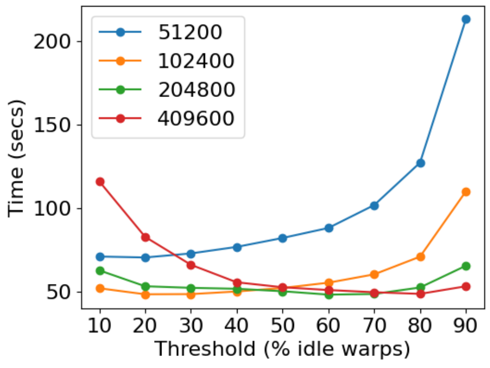
(d) ca-AstroPh, $k = 5$

Source: created by the author.

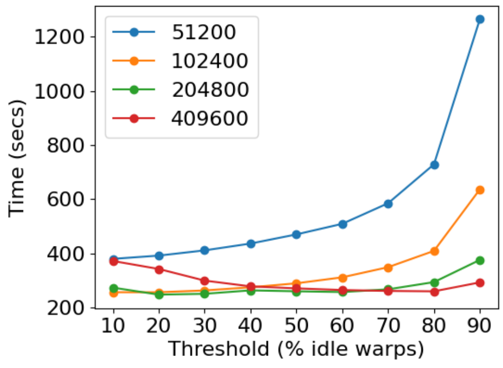
threshold, Figure 6.6 presents the load-balancing pattern of executions varying the threshold for both applications. We tested two opposite values of threshold (10% and 90%), and the best one was found in the sensitivity analysis of Figures 6.4 and 6.5 (30%). When the threshold is 10%, the average percentage of active warps is kept higher than the other thresholds throughout execution. However, increasing GPU occupancy does not necessarily result in better performance when the occupancy is higher than a certain minimum, achieved when up to 30% of threads are idle (around 28 active warps per SM). As seen in Figures 6.6(a) and 6.6(d), a side effect of decreasing the threshold is an increase in the number of calls to the load-balancing layer, which increases the overhead of the load-balancing layer throughout execution and deteriorates performance.

The threshold of 90% decreases the overhead of the load-balancing layer compared to the threshold of 10%, as fewer calls to workload redistribution are performed. However, this execution keeps an average amount of active warps lower than the minimum required to achieve efficiency, which also deteriorates performance. The threshold of 30% presents

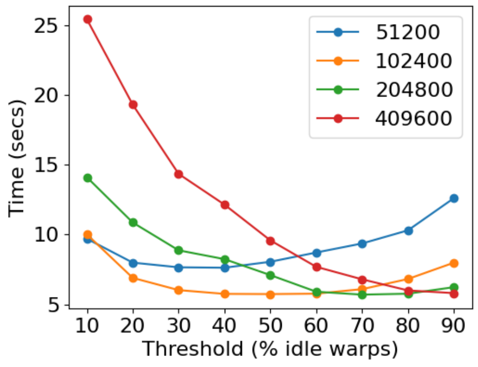
Figure 6.5: Impact of load-balancing threshold and number of threads to execution time (clique counting).



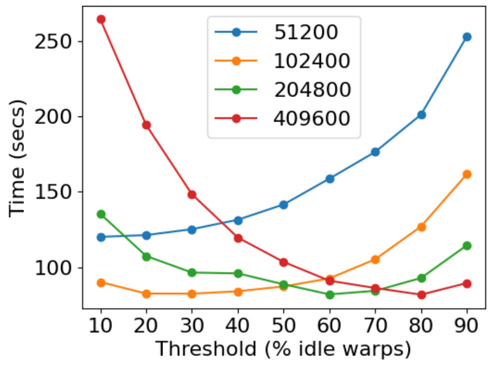
(a) ca-AstroPh, $k = 8$



(b) ca-AstroPh, $k = 9$



(c) DBLP, $k = 6$



(d) DBLP, $k = 7$

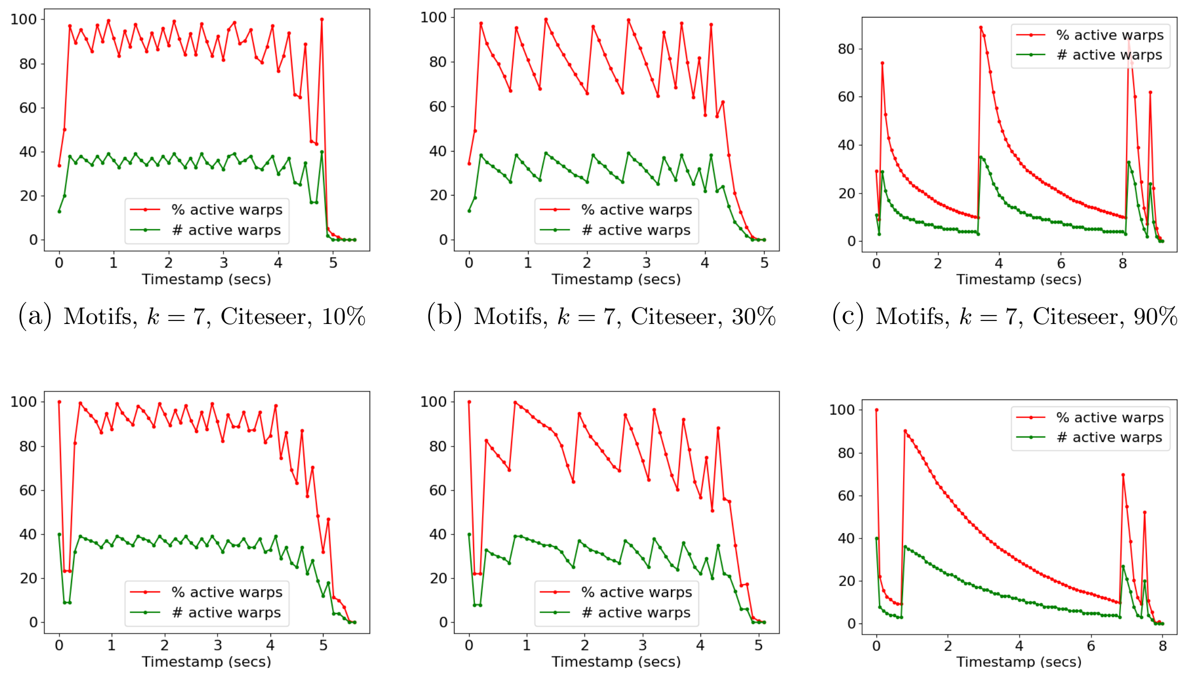
Source: created by the author.

the best tradeoff between the number of calls to the load-balancing layer and the minimum amount of active warps throughout execution.

6.3.2 Amount of Donations

Our novel load-balancing algorithm may assign more than one traversal per warp during the workload redistribution. Table 6.6 shows the execution time varying the number of donations per warp. An increase in this parameter results in improvements for all configurations up to 8 donations per warp. After that point, there are gains with 16 donations per warp for most configurations, but there are also few cases in which increasing the parameter leads to a minor performance penalty. Overall, the case with 16 donations

Figure 6.6: Rebalancing pattern with different thresholds.



(a) Motifs, $k = 7$, Citeseer, 10%
(b) Motifs, $k = 7$, Citeseer, 30%
(c) Motifs, $k = 7$, Citeseer, 90%
(d) Clique, $k = 6$, DBLP, 10%
(e) Clique, $k = 6$, DBLP, 30%
(f) Clique, $k = 6$, DBLP, 90%

Source: created by the author.

per warp is the best performing configuration. The number of donations per warp has also an impact (reduction) in the number of calls to the load-balancing layer. For instance, the execution of clique for ca-AstroPh and $k = 8$ spent 13% of the total execution time with load balancing with 1 donation, while the same execution with 16 donations spent only 3% with load-balancing.

App.	Dataset	k	Donations				
			1	8	16	32	64
Motifs	Citeseer	7	7.03	5.26	5.06	5.78	5.65
		8	108.29	97.03	96.95	97.79	98.88
	ca-AstroPh	5	147.98	129.21	126.78	126.67	128.23
		6	24328.37	23501.34	23625.05	23692.57	23700.19
	Mico	4	35.63	24.72	23.27	22.14	23.13
		5	7924.87	7641.68	7621.54	7599.79	7589.40
	DBLP	5	45.89	34.05	31.90	30.92	30.58
		6	2838.40	2645.50	2648.65	2649.30	2642.99
Clique	ca-AstroPh	10	1184.77	1144.80	1155.25	1157.83	1158.26
		11	5371.64	5271.42	5320.67	5347.48	5346.87
	Mico	5	49.78	34.38	33.30	34.45	35.22
		6	1453.21	1359.41	1354.69	1368.33	1364.97
	DBLP	7	86.04	69.84	70.75	71.93	72.22
		8	1177.41	1115.44	1122.30	1132.54	1140.21
	LiveJournal	4	81.46	43.57	34.56	29.92	26.50
		5	1439.97	658.52	602.53	576.59	565.73

Table 6.6: Execution time varying the job sizes.

6.3.3 Warp-centric vs Warp-centric with Load-balancing

Table 6.7 shows that the *DM_ WCLB* version attained speedups of up to 65× compared to *DM_ WC* (Motifs app, *Citeseer* dataset and $k = 8$). As the size of the enumerated subgraphs increases, work skewness grows because most subgraphs are extracted from denser regions of the graph associated with increasingly fewer vertices. At this point, load balancing becomes more effective. Hence, *DM_ WCLB* allowed the exploration of larger subgraphs for all datasets. Whenever the amount of work is insufficient to exhibit a substantial imbalance or to pay off the overhead of redistributing the load ($k \leq 4$ in small datasets), *DM_ WC* outperforms *DM_ WCLB*.

		System	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$	$k=12$
Motifs	Citeseer	DM_WC	0.0	0.04	0.91	18.09	334.32	9.66K	-	-	-	-
		DM_WCLB	0.11	0.12	0.24	0.67	5.06	96.95	-	-	-	-
	ca-Astr.	DM_WC	0.11	27.88	6.87K	-	-	-	-			
		DM_WCLB	0.25	1.47	126.78	23.63K	-	-	-	-	-	-
	Mico	DM_WC	1.24	768.10	-	-	-	-	-			
		DM_WCLB	0.47	23.27	7.62K	-	-	-	-	-	-	-
	DBLP	DM_WC	0.07	11.33	1.66K	-	-	-	-			
		DM_WCLB	0.13	1.11	31.9	2.65K	-	-	-	-	-	-
Clique	Citeseer	DM_WC	0.01	0.01	0.01	0.01	∅	∅	∅			
		DM_WCLB	0.11	0.12	0.12	0.13	∅	∅	∅			
	ca-Astr.	DM_WC	0.01	0.25	2.91	24.66	190.55	1.65K	12.27K			
		DM_WCLB	0.13	0.27	0.57	1.88	7.92	42.0	228.48	1.16K	5.32K	22.26K
	Mico	DM_WC	0.13	10.70	527.10	26.22K	-	-	-			
		DM_WCLB	0.3	1.27	33.31	1.35K	-	-	-	-	-	-
	DBLP	DM_WC	0.01	0.21	6.74	196.36	4.78K	-	-			
		DM_WCLB	0.13	0.29	0.78	5.17	70.76	1.12K	16.05K	-	-	-
	LiveJr.	DM_WC	14.01	219.65	6.17K	-	-	-	-			
		DM_WCLB	3.32	34.56	602.53	29.95K	-	-	-	-	-	-

∅: no valid subgraphs

Table 6.7: Comparative performance. Execution time (seconds) of *DM_ WC* and *DM_ WCLB*.

6.4 Comparison with State-of-the-art GPGPM Environments

This section compares our optimal DuMato GPU implementations (DM_WCLB) against four representative state-of-the-art subgraph enumeration systems: G2Miner [11] and Pangolin [12], both designed for GPU; Fractal [16] and Peregrine [28], both designed for parallel CPU machines. Table 6.8 shows the results. For each dataset and value of k, we emphasize in bold the best execution time(s). DuMato is more scalable and able to explore larger subgraphs than all the baselines within the same time limit, exploring subgraphs of up to 12 vertices. To the best of our knowledge, subgraphs of such size have not been explored by any other GPM system searching for exact outputs, showing that we can reduce the impacts of combinatorial explosion and improve scalability.

G2Miner is the state-of-the-art subgraph enumeration system for GPU. As it follows the pattern-aware paradigm, it must generate an execution plan for any query pattern before the execution, and its limitations become apparent when we increase the size of the subgraphs mined for the motif counting application. G2Miner has a hardcoded restriction concerning the number of patterns the motif counting application can process and is limited by $k = 4$. Our enumeration strategies are more scalable and can reach larger subgraph sizes.

G2Miner performs faster than DuMato for the clique counting application, as it represents a scenario using a widely studied single pattern (clique), which allows the creation of a custom efficient execution plan. However, two details must be pointed out in this scenario: even in its best-case scenario, G2Miner is not able to generate execution plans for cliques larger than 8 vertices (e.g., ca-Astroph and DBLP), thus it also presents scalability limitations concerning the subgraph size; if we change the pattern, G2Miner is not able to create execution plans on-the-fly (its code generator for GPU is not functional), while DuMato is not restricted by the desired pattern.

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 size of explored subgraphs is close to 5 vertices, limiting the applicability of GPM algorithms. Compared to Fractal, we obtain significant speedups in all executions with gains ranging from 12× to 78×. As the size of the explored subgraphs increases, the processing cost is higher, and DuMato exploits better GPU's massively parallel processing and achieves more significant gains.

Regarding Peregrine, DuMato is competitive for small values of explored subgraphs (up to 5 vertices), and shows speedups of up to 115x for larger explored subgraphs. Even in Peregrine's best case (clique application, which contains only one pattern), DuMato

		System	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$	$k=12$
Motifs	Citeseer	`DuMato`	0.11	0.12	0.24	**0.67**	**5.06**	**96.95**	-	-	-	-
		`G2Miner`	**0.01**	ERR	NS	NS	NS	NS	NS	NS	NS	NS
		`Pangolin`	**0.01**	**0.01**	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	**0.01**	**0.01**	**0.05**	3.47	537.66	-	-	NS	NS	NS
		`Fractal`	5.17	5.20	5.69	12.44	163.48	-	-	-	-	-
	ca-Astroph	`DuMato`	0.25	1.47	126.78	23.62K	-	-	-	-	-	-
		`G2Miner`	**0.01**	**0.05**	NS	NS	NS	NS	NS	NS	NS	NS
		`Pangolin`	**0.01**	0.21	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	**0.01**	0.57	132.90	52.80K	-	-	-	-	-	-
		`Fractal`	9.13	435.64	4.72K	-	-	-	-	-	-	-
	Mico	`DuMato`	0.47	23.27	7.62K	-	-	-	-	-	-	-
		`G2Miner`	**0.01**	**0.84**	NS	NS	NS	NS	NS	NS	NS	NS
		`Pangolin`	**0.01**	3.31	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	0.06	6.57	7.92K	-	-	-	-	-	-	-
		`Fractal`	16.43	474.46	-	-	-	-	-	-	-	-
	DBLP	`DuMato`	0.13	1.11	31.90	2.64K	-	-	-	-	-	-
		`G2Miner`	**0.01**	0.84	NS	NS	NS	NS	NS	NS	NS	NS
		`Pangolin`	**0.01**	**0.17**	INC	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	0.07	0.95	78.59	50.95K	-	-	-	-	-	-
		`Fractal`	14.33	37.62	1.43K	-	-	-	-	-		
Clique	Citeseer	`DuMato`	0.11	0.12	0.13	0.14	∅	∅	∅	∅	∅	∅
		`G2Miner`	**0.01**	**0.01**	**0.01**	**0.01**	∅	∅	∅	∅	∅	∅
		`Pangolin`	**0.01**	**0.01**	**0.01**	**0.01**	∅	∅	∅	∅	∅	∅
		`Peregrine`	**0.01**	0.03	0.02	0.02	∅	∅	∅	∅	∅	∅
		`Fractal`	4.84	4.83	4.75	4.81	∅	∅	∅	∅	∅	∅
	ca-Astroph	`DuMato`	0.13	0.27	0.57	1.88	7.92	42.0	**228.48**	**1.15K**	**5.32K**	**22.26K**
		`G2Miner`	**0.01**	**0.01**	**0.01**	**0.08**	0.68	**5.01**	NS	NS	NS	NS
		`Pangolin`	**0.01**	**0.01**	0.02	0.11	**0.61**	OOM	OOM	OOM	OOM	-
		`Peregrine`	**0.01**	0.10	0.83	6.38	43.56	272.42	1.55K	7.93K	36.26K	-
		`Fractal`	8.17	9.75	15.89	78.09	439.16	2.30K	12.89K	57.02K	-	-
	Mico	`DuMato`	0.30	1.27	33.31	1.35	49.49K	-	-	-	-	-
		`G2Miner`	**0.01**	**0.02**	**0.74**	**31.98**	**1.16K**	**39.58K**	NS	NS	NS	NS
		`Pangolin`	**0.01**	0.05	2.93	OOM	-	-	-	-	-	-
		`Peregrine`	0.09	1.81	82.67	3.66K	-	-	-	-	-	-
		`Fractal`	14.17	48.53	1.44K	56.72K	-	-	-	-	-	-
	DBLP	`DuMato`	0.13	0.29	0.78	5.17	70.76	1.12K	**16.05K**	-	-	-
		`G2Miner`	**0.01**	**0.01**	**0.02**	**0.37**	**8.16**	**148.23**	NS	NS	NS	NS
		`Pangolin`	**0.01**	0.01	0.03	0.50	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	0.11	0.16	1.36	25.92	531.88	9.35K	-	-	-	-
		`Fractal`	13.44	14.32	22.72	186.97	2.52K	35.51K	-	-		
	LiveJournal	`DuMato`	3.31	34.56	602.53	29.95K	-	-	-	-	-	-
		`G2Miner`	0.02	**0.21**	**6.39**	**318.98**	**14.95K**	-	NS	NS	NS	NS
		`Pangolin`	**0.01**	0.53	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
		`Peregrine`	3.91	26.66	1.06K	64.74K	-	-	-	-	-	-
		`Fractal`	394.85	901.05	16.06K	-	-	-	-	-	-	-

OOM: out-of-memory; *INC*: incomplete results;
∅: no valid subgraphs; *ERR*: execution error; *NS*: not supported.

Table 6.8: Comparative performance. Execution time (seconds) of DuMato and baselines (GPU and CPU).

can deliver consistent speedups. We achieve more expressive gains in the motif counting application for larger values of k, which may be explained by the inherent characteristics of the pattern-aware enumeration of Peregrine. As we increase the size of explored subgraphs, the number of valid patterns and exploration plans grows exponentially, incurring two aspects that impact Peregrine's 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.

In summary, this chapter shows that our proposed subgraph enumeration strategies improved the efficient use of GPU's computing power compared to state-of-the-art methods for subgraph enumeration on GPU. The DFS-wide subgraph exploration presents an affordable data structure to store the intermediate enumeration data and opportunities for execution parallelism and regular memory requests. The warp-centric enumeration phases provide a more regular execution throughout the enumeration execution workflow, maximizing GPU's SIMT efficiency. Our load-balancing layer provides a low-cost workload redistribution scheme that mitigates the inherent load imbalance of parallel subgraph enumeration. Implementing and evaluating our novel subgraph enumeration strategies in the DuMato system confirms our hypothesis that novel subgraph enumeration strategies were needed to improve the efficient use of GPU's computing power for this problem.

Chapter 7

Final Remarks

In this work, we propose novel strategies to mitigate the main challenges for efficient subgraph enumeration on GPUs: irregularity, which limits the use of GPU's massive parallelism and HBRAM; and combinatorial explosion, which creates high memory demands and limits the scalability of GPM algorithms. 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. We also proposed and evaluated an implementation with warp virtualization, providing new opportunities for fine-grained parallelism of subgraph enumeration on GPU.

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 redistributes enumeration jobs considering the weight of each warp. Jobs are extracted from the heaviest warps and warps receive several jobs in a workload redistribution step, increasing the GPU occupancy and reducing the calls to the load-balancing layer.

All of our optimization strategies were implemented in an open-source system called DuMato, which provides a high-level functional API for the design and execution of GPM algorithms on GPU. This system contributes to widen the efficient use of GPM algorithms even for users unfamiliar with GPUs, thus increasing research opportunities to take advantage of these algorithms in new scenarios.

7.1 Limitations and Future Work

There are a few fronts of improvement in this work. Warp virtualization did not bring performance gains in the warp-centric execution workflow when virtual warps receive different traversals. The inherent load imbalance of subgraph enumeration associated with the conservative GPU scheduling of virtual warps reduces the massive parallelism of this virtualization approach. As a next step, we plan to use warp virtualization during the visitation of adjacency lists of the same traversal rather than assigning different traversals to different virtual warps. This way, we could ensure a lockstep execution between different virtual warps and reduce the amount of idle virtual warps within the same physical warp.

Our load-balancing mechanism uses the CPU to check when load-balancing is needed and to perform workload redistribution. We could extend our *write* primitive to help load-balancing. This primitive is responsible for writing the intermediate states, and it could redirect a portion of the extensions to a queue of jobs consumed by the CPU asynchronously. In this strategy, the CPU would not stop GPU execution to perform workload redistribution, thus reducing the overhead of the load-balancing layer.

The implementation of DuMato is currently pattern-oblivious, which gives us the advantage of enumerating larger subgraphs without the need for custom exploration plans. However, our experiments showed that the pattern-aware paradigm may be more efficient for a scenario concerning a limited amount of smaller patterns. As DuMato proposes a general-purpose API, we also plan to extend its primitives to implement the pattern-aware enumeration paradigm. This way, we will have a subgraph enumeration system that takes advantage of both paradigms, allowing a detailed comparison using the same software environment and giving new insights into the suitability of each enumeration paradigm.

The size of the enumerated subgraphs does not limit the execution of our strategies using DuMato. However, due to the exponential growth in the number of subgraphs as we increase k, the complete execution of subgraph enumeration may take a long time to finish even in an optimized environment. We plan to propose an interface to DuMato to allow the streaming of the visited subgraphs with k vertices, thus allowing the collection of results on the fly. As the pattern size does not limit us and we use a DFS-like traversal strategy, we can visit larger subgraphs and produce results gradually.

Appendix A

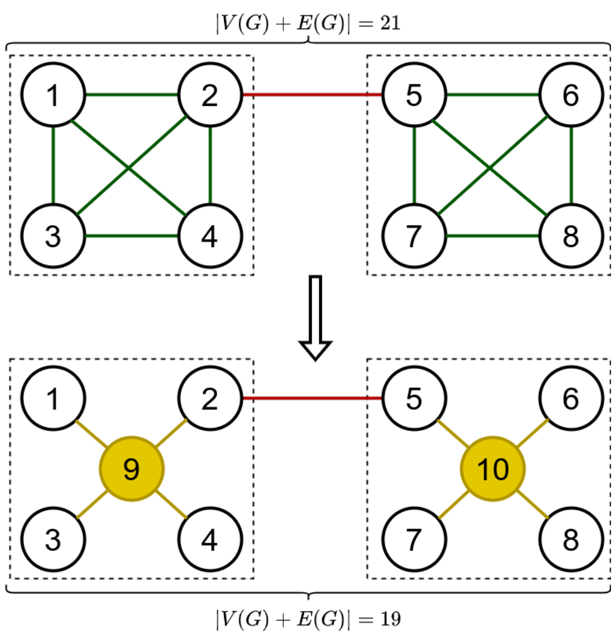
Use Case: Graph Compression

Graph compression is a research area dedicated to finding a shorter representation of a graph G. The reason for compressing a graph varies depending on the context: reduce the storage space, reduce I/O cost, improve the overall memory efficiency, and so forth. Besides, some graph compression techniques may be applied to any graph while others are domain-specific [5]. Our goal in this appendix is to propose a graph compression algorithm using DuMato. We want to demonstrate the applicability of DuMato in an end-to-end application and illustrate opportunities to exploit when having a subgraph enumeration system capable of visiting larger subgraphs.

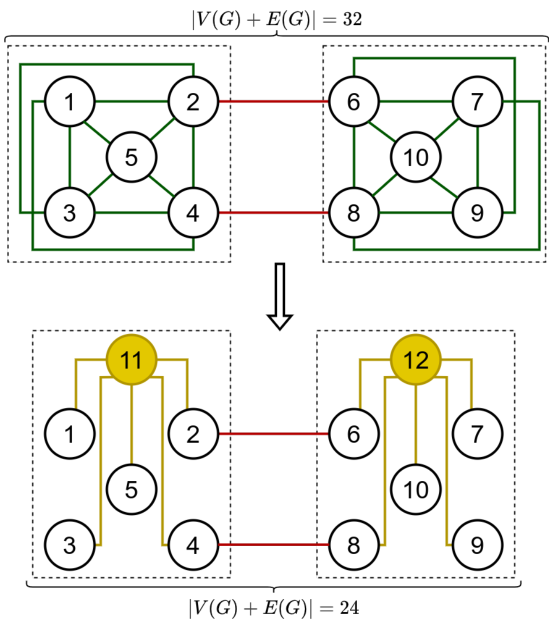
Graph compression algorithms replace parts of the graph with other representations according to a criterion. Our graph compression algorithm uses DuMato to enumerate all cliques of a size k and replace these cliques using a lossless shorter representation, depicted by Figure A.1. For each clique in the graph, we insert a new vertex representing that clique and connect the vertices belonging to that clique to this new vertex. For example, in Figure A.1(a), there are two cliques with four vertices, and we create the vertices 9 and 10 to represent them, along with the edges connecting the vertices that belonged to the corresponding cliques. In Figure A.1(b), we create the vertices 11 and 12 to represent the cliques with 5 vertices. Assuming the cost to store subgraphs is the sum of its vertices and edges, the compression rate using cliques with four vertices is 9%, while the compression rate using cliques with five vertices is 25%. As we increase the size of the cliques, we reduce the overhead caused by adding the new vertex and edges.

A similar technique has been used by other paper [49], but with a difference: they replace entire maximal cliques, thus ignoring the chances of these cliques sharing edges with other cliques can also reduce the size of the subgraph. Instead, when we replace a clique, we keep the edges that are shared with other cliques. We enumerate all possible cliques with k vertices and rank them according to their capacity to reduce the size of the graph. If a clique shares edges with other cliques, we remove only the edges that are not shared, thus keeping other cliques that may also be used to compress the graph later. Figure A.2 depicts our replacement strategy used for compression. Dotted edges represent edges shared between cliques. When the clique surrounded with a dashed line is replaced using the mechanism described in Figure A.1, the shared edges are kept, thus

Figure A.1: Lossless graph compression using cliques.



(a) $k = 4$.

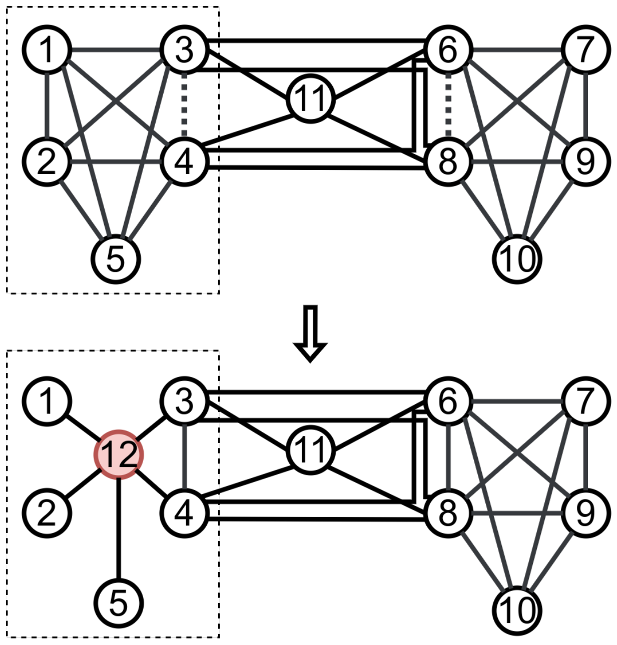


(b) $k = 5$.

Source: created by the author.

keeping the structure of the graph and allowing the clique in the middle to be used for compression in the future.

Figure A.2: Clique compression without removing shared edges.



Source: created by the author.

Given a clique C, $rank(C)$ indicates the compression capability of C and is calculated as:

$$rank(C) = ((k \times (k-1))/2) - (shared\_edges(C) + k + 1) \tag{A.1}$$

In the first part of the Equation A, $((k \times (k-1))/2)$ represents the set of all edges of a clique with k vertices. In an ideal scenario, all these edges should be removed when a clique is used for compression. The second part of the equation represents the overhead of the compression. We do not remove the shared edges, and we also have to add k edges and one vertex to create the compressed graph, thus reducing the ranking of the clique. The fewer shared edges a clique contains, the more compressive it will be. As we enumerate all possible cliques with k vertices, we can rank them to maximize their compression capabilities.

Algorithm 19 uses DuMato API to implement the compression algorithm that uses cliques with k vertices. The output is the compression score, which represents how many edges/vertices of the graph can be removed if cliques with k vertices are used to compress the graph. The left column represent the GPU code, which is written using DuMato API. This code is almost the same code *clique_counting* code depicted in Algorithm 9, but with one difference: in line 10 we store the cliques rather than counting. This is necessary to calculate the compression scores of each clique later on CPU.

The right column represents the CPU code executed after all the enumeration workflow finishes. Lines 15-19 count the times each edge appears in each enumerated clique. Lines 22-29 rank each clique based on the number of shared edges. Lines 32-37 calculate the compression score using only the cliques that compress the original graph (those with a rank greater than 1). If we wanted to compress the graph, we could replace lines 35-36 with the code to create the compressed graph.

Table A.1 shows the number of compressing cliques (those with rank ≥ 1) and the number of edges that can be removed if we apply our lossless compression using all enumerated cliques of size k. Note that as we increase the size of visited subgraphs, the compression using cliques becomes more effective. Besides, note that we used cliques up to 9 vertices, which is not possible if we use the state-of-the-art subgraph enumeration systems for GPU.

```
// GPU code
void clique_counting(TE){
 while(control(TE)){
   if(!extend(TE, 0, 1)){
     u ← TE[TE.len − 1].id;
     filter(TE, &lower_than , [u]);
     filter(TE, &is_clique, []);
   }
   if(TE.len == k − 1)
     aggregate_store(TE);
   move(TE, false);
 }
}
```

```
//CPU code
$edges \leftarrow \{\}$
foreach $clique \in cliques$ do
    foreach $edge \in clique.edges$ do
        $edges[edge] + +$;
    end
end

$rank \leftarrow \{\}$
foreach $clique \in cliques$ do
    $rank[clique] \leftarrow (k * (k - 1))/2$;
    $rank[clique] - = (k + 1)$;
    foreach $edge \in clique.edges$ do
        $if(edges[edge] > 1)$
          $rank[clique] - -$;
    end
end

$sort(rank, descending)$;
$score \leftarrow 0$;
foreach $clique \in cliques$ do
      $if(rank[clique] >= 1)$
        $score + = rank[clique]$;
end
return $score$;
```

Algorithm 19: Compression algorithm.

k	# cliques	# compressing cliques	# removed edges
4	9580415	412	412
5	64997961	1134	2250
6	400401488	1710	5094
7	2218947958	2126	8548
8	11088038672	2442	12276
9	50170510922	2658	15384

Table A.1: Lossless compression of ca-AstroPh using the cliques.

References

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad 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, pages 61:1–61:12, Piscataway, NJ, USA, 2016. IEEE Press.

[2] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Parallel k-clique counting on gpus. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[3] AMD. AMD Homepage. https://www.amd.com, 2022.

[4] Albert-László Barabási. Scale-free networks: A decade and beyond. *Science*, 325(5939):412–413, 2009.

[5] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations, 2019.

[6] Malay Bhattacharyya and Sanghamitra Bandyopadhyay. Mining the largest quasi-clique in human protein interactome. In *2009 International Conference on Adaptive and Intelligent Systems*, pages 194–199, Sep. 2009.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1199–1214, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Advances in Knowledge Discovery and Data Mining*, pages 858–863, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *GH '08*, GH '08, 2008.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the*

Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Xuhao Chen and Arvind. Efficient and scalable graph pattern mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 857–877, Carlsbad, CA, July 2022. USENIX Association.

[12] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(10):1190–1205, April 2020.

[13] Sarvenaz Choobdar, Pedro Ribeiro, Sylwia Bugla, and Fernando Silva. Comparison of co-authorship networks across scientific fields using motifs. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 147–152, 2012.

[14] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs*. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 589–598, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[15] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *WWW '18*, WWW '18, 2018.

[16] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1357–1374, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Alexandra Duma and Alexandru Topirceanu. A network motif based approach for classifying online social networks. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 311–315, 2014.

[18] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, mar 2014.

[19] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM J. Exp. Algorithmics*, 18, November 2013.

[20] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. Sumpa: Efficient pattern-centric graph mining with pattern abstraction. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–330, 2021.

[21] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1067–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD '20*, SIGMOD, 2020.

[23] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon 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, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *PPoPP '11*, PPoPP '11, 2011.

[25] Bryan Hooi, Kijung Shin, Hemank Lamba, and Christos Faloutsos. Telltail: Fast scoring and detection of dense subgraphs. In *AAAI '20*, AAAI '20, 2020.

[26] Yang Hu, Hang Liu, and H. Howie 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] Eslam Hussein, Abdurrahman Ghanem, Vinicius Vitor dos Santos Dias, Carlos H.C. Teixeira, Ghadeer AbuOda, Marco Serafini, Georgos Siganos, Gianmarco De Francisci Morales, Ashraf Aboulnaga, and Mohammed Zaki. Graph data mining with arabesque. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1647–1650, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages YYYY–YYYY, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. 2014.

[30] Terence Kelly. Compressed sparse row format for representing graphs. In *Compressed Sparse Row Format for Representing Graphs*, USENIX Winter 2020, 2020.

[31] Robert Kessl, Nilothpal Talukder, Pranay Anchuri, and Mohammed Zaki. Parallel graph mining with gpus. In Wei Fan, Albert Bifet, Qiang Yang, and Philip S. Yu, editors, *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, volume 36 of *Proceedings of Machine Learning Research*, pages 1–16, New York, New York, USA, 24 Aug 2014. PMLR.

[32] Nikhil S. Ketkar, Lawrence B. Holder, and Diane J. Cook. Subdue: Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, OSDM '05, page 71–76, New York, NY, USA, 2005. Association for Computing Machinery.

[33] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217–228, nov 2016.

[35] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 2007.

[36] W. Lin, X. Xiao, X. Xie, and X. Li. Network motif discovery: A gpu approach. In *2015 IEEE 31st International Conference on Data Engineering*, pages 831–842, Address, 2015. Publisher.

[37] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *ECMLPKDD '08*, ECMLPKDD '08, 2008.

[38] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. Community detection in complex networks via clique conductance. *Scientific Reports.*, April 2018.

[39] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37, jun 2021.

[40] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 509–523, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 2014.

[42] Md Moniruzzaman Monir and Ahmet Erdem Sarıyüce. Using large cliques for hierarchical dense subgraph discovery. In Sriram Chellappan, Kim-Kwang Raymond Choo, and NhatHai Phan, editors, *Computational Data and Social Networks*, pages 179–192, Cham, 2020. Springer International Publishing.

[43] NVIDIA. Volta Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[44] NVIDIA. NVIDIA Homepage. https://www.nvidia.com, 2022.

[45] NVIDIA. Profiler User Guides, howpublished="https://docs.nvidia.com/cuda/profiler-users-guide/", 2023.

[46] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, apr 2016.

[47] Pedro Ribeiro and Fernando Silva. G-tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov.*, 28(2):337–377, March 2014.

[48] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[49] Ryan A. Rossi and R. Zhou. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data*, 5, 2018.

[50] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong 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.

[51] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[52] 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. In *Euro-Par 2014 Parallel Processing*, pages 451–462, Cham, 2014. Springer International Publishing.

[53] Olaf Sporns and Rolf Kötter. Motifs in brain networks. *PLOS Biology*, 2(11):null, 10 2004.

[54] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery.

[55] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. Fast subgraph matching on large graphs using graphics processors. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications*, pages 299–315, Cham, 2015. Springer International Publishing.

[56] 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. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 763–782, USA, 2018. USENIX Association.

[57] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: Gpu graph analytics. *ACM Trans. Parallel Comput.*, 4(1), aug 2017.

[58] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, and John 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)*, pages 1369–1380, 2020.

[59] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *MDS '12*, MDS '12, 2012.

[60] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Machine Learning: Fundamental Concepts and Algorithms.* Cambridge University Press, Address, March 2020.

[61] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An efficient out-of-core graph mining system on a single machine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 673–684, 2020.

[62] Feng Zhou and Fernando De la Torre. Deformable graph matching. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2922–2929, 2013.