ENHANCEMENTS IN HIGH PERFORMANCE SUBGRAPH ENUMERATION ON GRAPHICS PROCESSORS

Draft of July 7, 2022 at 12 : 04

BY

SAMIRAN KAWTIKWAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Industrial and Systems Enterprise Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Advisers:

Professor Rakesh Nagi
Professor Wen-mei Hwu

ABSTRACT

Subgraph enumeration is an important problem in graph theory with wide range of applications. Being NP-complete, this problem needs efficient implementations and smart heuristics to be practical for large sized instances. This problem has huge scope for parallelism, there are many existing solutions in the multi-core and distributed computing community. Graphics Processing Units (GPUs) offer massive parallelism and GPU based solutions outperform the multi-core implementations.

Most GPU solutions use Breadth First Traversal to utilize underlying parallelism and impose expensive restrictions on hardware due to huge memory requirements. PARSEC (Parallel Subgraph Enumeration and Counter) [1] is the first GPU based solution that uses depth first search and performs in-memory subgraph enumeration. In this thesis, PARSEC is improved with insights from traditional sequential solutions and smart GPU implementations.

The performance of subgraph Enumeration is limited by number of intersection operations. To tackle this, a smart preprocessing technique was developed that detects scope for intersection reuse to reduce the number of intersections by up to 3.87×. A 2 phase pruning technique was developed which shrinks search space to further reduce the number of intersections by up to 6.6×. An in depth analysis of PARSEC was conducted to discover severe load imbalance which limits performance, a hybrid parallelization scheme was developed that improves the load balance by up to 14×. Altogether, these improvements provide a geometric mean time speedup up to 4.6× across data graphs and up to 3.7× across all queries. [SK: Is this overselling?]

To Aai, Baba and Tai for their unconditional love and support.

ACKNOWLEDGMENTS

I express my deepest gratitude to my parents for their endless love and support, for their hardships and sacrifices enabling me to pursue my dreams.

I would like to thank my advisor Professor Rakesh Nagi for great mentorship and support. His energy and modesty has been a great source of motivation and humility. I would like to extend my sincerest acknowledgments to Professor Wen-mei Hwu for his excellent insights and Dr. Jinjun Xiong for his support and interest. [SK: Does this sound okay?] I am grateful to have found such great mentors to work with.

A special thanks to my colleagues at the C^{3}SR group: Mohammad, Yen-Hsiang and Vibhor for their excellent help and support. A huge appreciation to IBM C^{3}SR for partially funding my appointment and to the ISE admin staff especially Lauren Redman for being so helpful and accommodating through the pandemic.

My gratitude is also extended to my previous mentors: Professor Anand Dev Gupta and Dr. Varun Ramamohan for inspiring me to pursue research.

Lastly, I would also thank my friends in Urbana Champaign for all the fun and homely environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BFS	Breadth First Search
CPU	Central Processing Unit
DFS	Depth First Search
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multi Data
SM	Streaming Multiprocessor
HPC	High Performance Computing
FLOPS	Floating Point Operations Per Second

CHAPTER 1

INTRODUCTION

Graphs are widely used to represent complex information such as social networks, chemical compounds, World Wide Web, logistic networks, bioinformatics, etc. They are used due to their unique ability to efficiently represent complicated relational information. Graph algorithms have applications in diverse areas like molecular engineering, social sciences, operations research, computer science and many more. In mathematics, Graph theory is a field of study of graphs. Formally, a graph G is defined as a collection of vertices V and edges $E \in V \times V$ represented as $G = (V, E)$. Where vertices represent entities and an edge represents relation between a pair of entities.

Subgraph Enumeration is a fundamental problem in graph theory that aims at finding all instances of a given query graph G_q in a larger data graph G_d. Formally, given a query graph G_q and data graph G_d, the objective is to find all instances of subgraphs of G_d that are *isomorphic* to G_q. Figure 1.1 illustrates an instance of subgraph enumeration.

Isomorphic graphs are of interest since they represent similarity between inherent topology of the graph. For graphs G_1 and G_2 graph isomorphism is a bijection between the vertex sets $\mathcal{V}(G_1)$ and $\mathcal{V}(G_2)$ $f : \mathcal{V}(G_1) \rightarrow \mathcal{V}(G_2)$ with the condition: *any two vertices u and v of G_1 are adjacent in G_1 if and only if $f(u)$ and $f(v)$ are adjacent in G_2*. Mathematically, for G_1 and G_2 to be isomorphic: $(u, v) \in \mathcal{E}(G_1) \leftrightarrow (f(u), f(v)) \in \mathcal{E}(G_2)$. Where $\mathcal{E}(.)$ represents the edge set of a graph. In this work we focus on undirected query graphs and data graphs. Subgraph enumeration entails the subgraph isomorphism problem which is known to be NP-complete [2].

Subgraph Enumeration has wide range of applications. For example, it can be used for plagiarism detection [3]. It is used for network motif counting [4], comparing similarity between large graphs [5], designing bioinformatics networks [6], chemical target structure synthesis [7], analyzing insights in recommendation networks [8], etc.

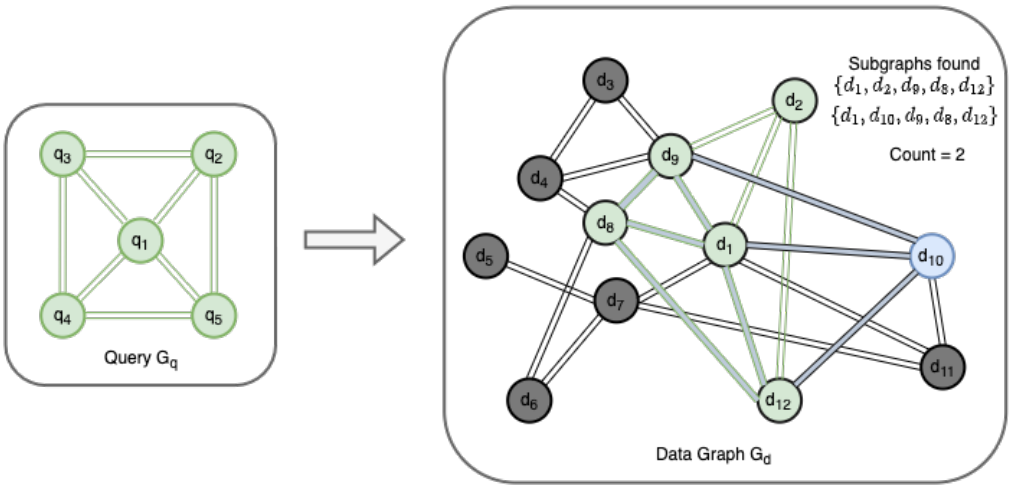


Figure 1.1: Subgraph Enumeration Example

With the advent of *Internet Of Things* (IOT) and *Information Technology* (IT), the sizes of graphs representing underlying data are substantially bigger and pose significant challenges for data scalability. Increasing interests in graph analytics and improvements to computational hardware over the last two decades, in particularly CPUs and GPUs have helped develop practical solutions to this computationally challenging problem. Though, the existing solutions don't scale with increasing template size due to issues like memory requirement, computational time, and load imbalance.

This work is focused towards improving the existing state-of-the-art solution for subgraph enumeration: PARSEC, developed by [1]. A series of enhancements were developed to improve its computational performance which are discussed at length in later sections. These enhancements include:

1. Smart preprocessing techniques to detect intersection reuse and reduce computation.

2. 2-Phase strategy to improve symmetry breaking effectiveness.

3. Hybrid parallelism for better load balance.

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview of Graphics Processing Units and graph storage formats.

Chapter 3 gives a Literature Review that describes related work and defines mathematical notation used throughout this thesis.

Chapter 4 illustrates basic techniques used in the subgraph enumeration algorithm with a runnning example.

Chapter 5 describes aforementioned enhancements in detail and analyzes their individual contribution to performance.

Lastly, Chapters 6 - 7 evaluate combined improvement due to these enhancements and wrap up the thesis with concluding remarks and scope for further work.

CHAPTER 2

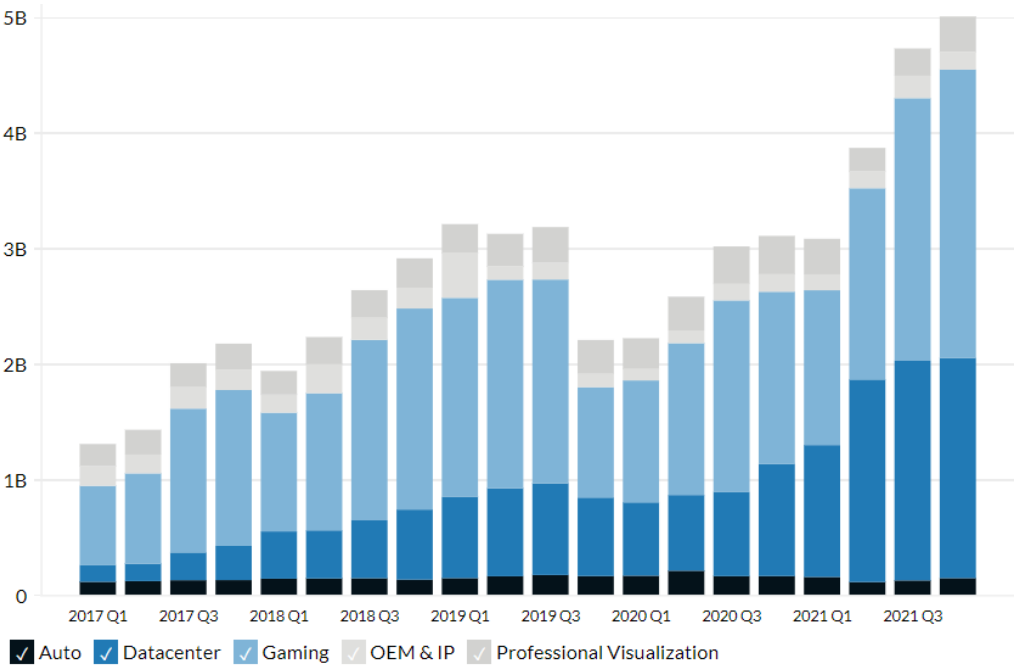
BACKGROUND

2.1 GPU Overview

2.1.1 History and Motivation

In the world of computational hardware, microprocessors based on single processing cores (traditional CPUs) have driven rapid performance improvements in floating-point operations per second (FLOPS) for nearly 2 decades ($1985 - 2003$). These microprocessors were capable of performing in the range of Giga-FLOPS (10^9) on desktops and Tera-FLOPS (10^{12}) on data centers [9]. This improvement drive flattened due to fundamental problems such as thermal stability, energy density, and quantum effects. Since then, all microprocessors have adapted architecture with multiple processing cores and thread virtualization to keep up with the improvements expected by the market. Naturally, it needed significant change in software to keep up with the reported performance and increasing computational requirements [10].

High-Definition Graphics demand from the gaming industry was one of the biggest drivers for such high-performance hardware. The performance demanded here is throughput oriented i.e., generating maximum frames per second while the computational requirement for each pixel is sufficiently low. Some chip manufacturers like NVIDIA took a different approach of multi-thread hardware to cater to these requirements [11]. The hardware designed with this philosophy had threads capable of performing simple computational tasks with relatively lower clock speed to give an overall performance in Tera-FLOPS range. On top of that, the hardware gave more performance per watt which made it possible to add to a desktop machine [12]. Modern GPUs have also proven to be fast and cost-effective for deep learning and image processing tasks contributing to many advances in fields like computer vision



Source: Business Quant [13]

Figure 2.1: Nvidia Revenue by End Market

and artificial intelligence. A modern data center GPU, for example (NVIDIA A100), now has more than 100,000 concurrent threads. These threads execute in many simple pipelines to give around 10 times faster raw performance than state-of-the-art CPUs [9].

Figure 2.1 shows that NVIDIA GPU data center market has improved significantly in recent years to become almost equivalent to the already huge gaming industry. This has fueled the hardware and software development of current server-grade GPUs which has enabled huge computational potential for deep learning. While HPC hardware improves at such rapid pace, software solutions for graph theory problems still need to move from the *multi core* paradigm to *multi thread*.

2.1.2 Heterogeneous Programming Environment

GPU is a throughput oriented computational hardware. A single thread of a GPU is significantly slower than a CPU. Hence, working with it for inherently sequential tasks can incur huge latencies. Since many basic programming tasks are inherently sequential and the whole operating system

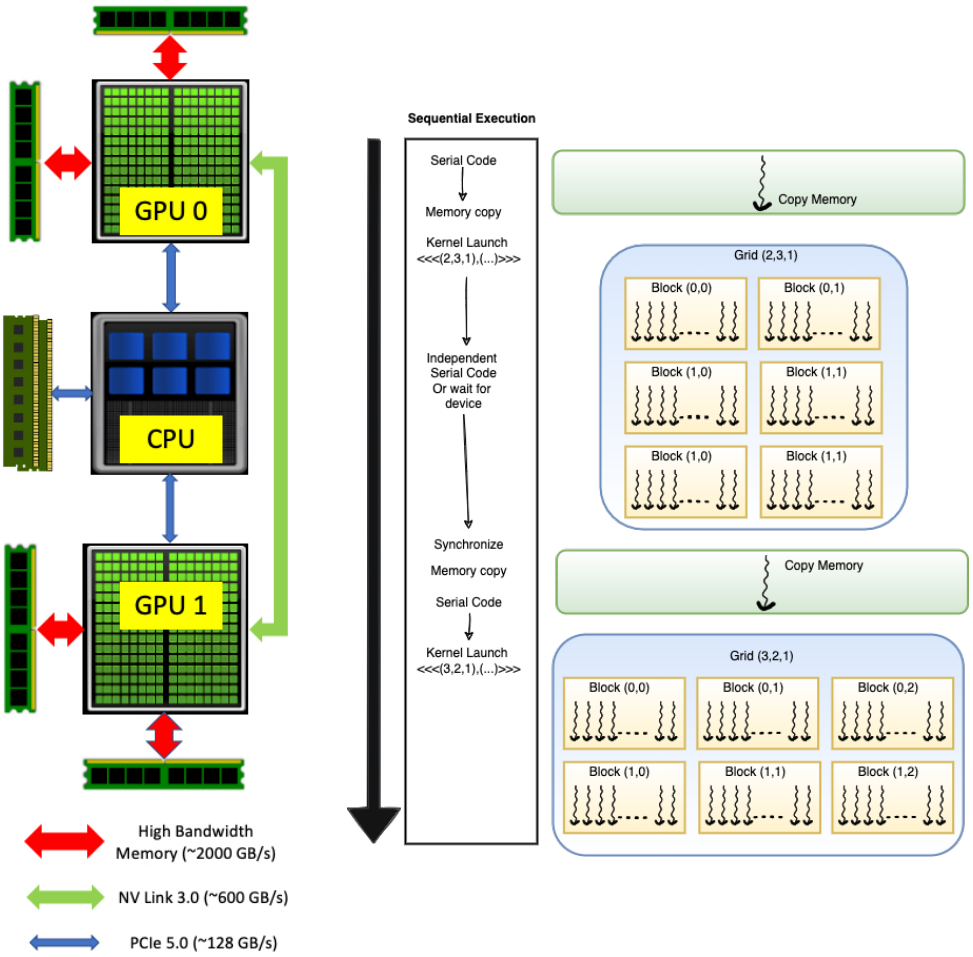


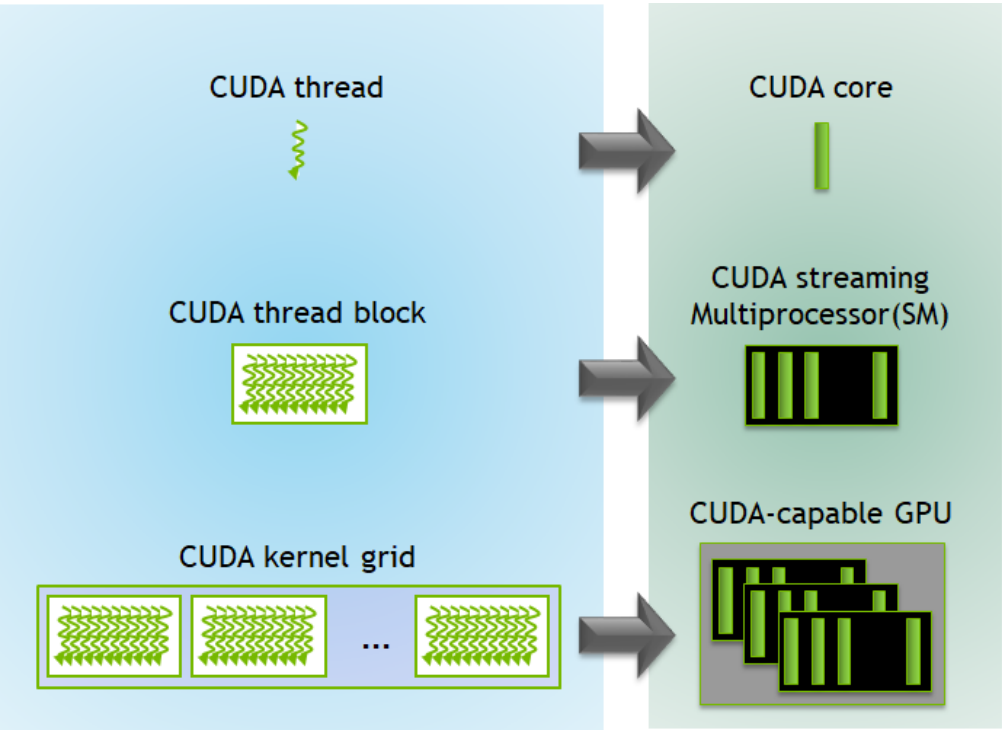
Figure 2.2: CUDA Heterogeneous Programming Model

environment is developed for latency optimized processors (CPUs), GPUs work better off being used as a supplemental piece of hardware along with CPUs for computationally intensive tasks. Hence, GPUs are mostly used in a Heterogeneous programming environment with CPU (host) executing the governing code while offloading computationally intensive tasks as kernels to GPUs (device). Note that one host can be connected to multiple devices. Since the memory of device and host are different, the input and output data need to be copied before and after kernel execution. Figure 2.2 shows the high level hardware and software architecture. GPUs can communicate with CPU via PCIe (Peripheral Component Interconnect express) bus and with each other via NVLink ™. The memory bandwidth of PCIe is the lowest and hence it is in best interests to avoid high data transfers between GPU and CPU. Native memory of the GPU or the VRAM, is High Bandwidth Memory

(HBM) with the latest generation having bandwidth of around 2000 GB/sec. The Nvidia proprietary GPU Interconnect with 12 links per GPU can provide up to 600 GB/sec. While the industry standard PCIe bus gen 5.0 with 32 links can provide up to 128 GB/sec of bandwidth.

2.1.3 Thread Hierarchy

As discussed in Section 2.1.1 GPUs have *multi thread* architecture with hundreds of thousands of threads. To have such massive hardware parallelism threads have to collaborate on usage of different resources for ex. *cuda cores*. Since all threads do not share these resources, it is important to understand thread Hierarchy in GPUs.



Source: Nvidia Developer Blog [14]

Figure 2.3: Thread Hierarchy

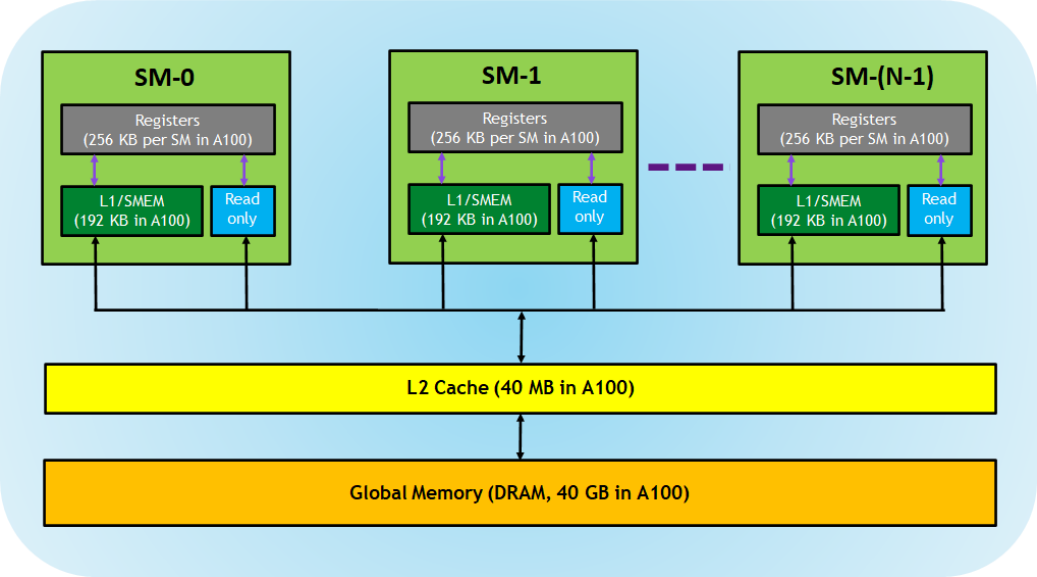
A thread is a single sequential flow of control within a program. A block is a chunk of threads while a grid is a chunk of blocks. Since threads are virtual they need to be optimally assigned hardware resources to perform tasks. Figure 2.3 shows how these virtual entities are mapped to the hardware. As

discussed a thread utilizes a *cuda core* for performing computation. A block of threads resides on a *Streaming Multiprocessor* (SM), which is a hardware block that consists of computational resources like *cuda cores, memory controllers, tensor cores*, etc. Thread block resides on an SM for the entirety of its execution. Hence, multiple threads in a block can utilize the same core. Multiple blocks can reside on an SM simultaneously within limits. GPU has multiple SMs and hence many blocks can concurrently reside on the GPU during a context. If a grid requests more blocks than that can concurrently reside on the GPU, their execution is serialized. Nvidia's latest server-grade GPU A100 has 108 SMs with a maximum of 32 blocks or 2048 threads residing per SM which means the maximum number of concurrent threads is 221,184. While on the hardware side, A100 has 6912 *cuda cores*, i.e., 64 cores per SM.

To add further, the thread instructions are processed in a Single Instruction Multi Data (SIMD) model. This SIMD chunk of threads is allocated resources at once, this chunk is called *warp*. If threads within a warp demand different operations then all threads within that warp perform all operations with output of undesired threads being ignored. This causes a slowdown in execution and is referred as *warp divergence*. Hence, algorithms and implementations need to be designed in such a way that *warp divergence* is minimized. While the size of thread block and grid are configurable by the programmer, warp size is a fixed architectural parameter.

2.1.4 Memory Hierarchy

GPU needs high performance memory system as it can be easily stressed due to the underlying massive parallelism. If the available memory bandwidth is overwhelmed, threads need to wait for data to be able to perform computation even though some processing cores are free. This results in poor utilization and overall slower performance. Even though GPU memory is significantly faster than conventional RAM, it can still limit application performance. This problem is alleviated by building memory caches on the chip die itself, so accessing them is significantly faster (around 100 - 1000 times) but they have size restrictions. Modern compilers identify repeated memory accesses and can aggressively cache to reduce memory stress. The



Source: Nvidia Developer Blog [14]

Figure 2.4: GPU Memory Hierarchy

memory bandwidth problem is not significant for CPUs as they have huge cache banks per thread core. But for GPUs the cache per thread (L1 cache) is very restrictive.

GPU memory architects have tried to solve this problem by adding hierarchical memory banks. There are 3 types of memory banks on a GPU namely *Global*, *Shared* and *Register* memory. Figure 2.4 shows the memory hierarchy, as name suggests global memory is accessible by all threads, it is equivalent to the Random Access Memory (RAM) on CPUs and is also sometimes referred as Video RAM (VRAM) or Device RAM (DRAM). Shared memory is accessible by all threads in a thread block. Register memory is private to each thread. The size limitations on these memory banks are based on SM architecture and hardware. Programmers are also equipped with read-only constant memory (not shown in the Figure 2.4) which is accessible by all threads. Unknown to the programmer, compilers can also utilize these memory banks by caching in register and shared memory banks if repeated accesses to global memory are detected. It is referred as L1 cache. Equivalent to CPU cache there is also an L2 cache for use by the compiler only and accessible to all threads.

As discussed, memory utilization plays a significant role in code perfor-

mance, a program whose execution time is restricted by memory is called memory bound. Memory bound programs do not scale well with increasing number of devices as distributing computation to other workers does no benefit and further increases latency due to communication. Some common techniques to reduce memory pressure are designing algorithms capable of using shared and register memory, consecutive threads accessing nearby memory and reducing contention for atomic operations on global memory.

2.1.5 Block Scheduling

As discussed in Section 2.1.3 many blocks can concurrently reside on a GPU.

Generally kernels are launched with grid sizes much higher than that can concurrently reside on a GPU, so the block execution needs to be serialized once maximum number of blocks that can occupy the GPU are scheduled, the blocks waiting after initial allocation are allocated resources as they become available (i.e. a scheduled block terminates). This is done as blocks are non-preemptive. That is, ones allocated resources the block is never unallocated till its completion. This scheduling provides scope clever memory utilization as memory need not be allocated for whole grid but only the blocks that can run concurrently. This concept of memory management is called persistent memory, using it can help reduce memory requirements.

Since blocks are preemptive, it is important to design the parallelism in such a way that all blocks have nearly similar amount of work. If not, blocks having higher amount of work should be scheduled earlier to reduce make span. In order to have better control over work allocation to a block, programmers often launch kernels with grid size equivalent to that can concurrently reside. These blocks then can choose from a common list of tasks in global memory, this approach is known as work stealing. Work stealing makes sure that work is allocated to a worker as soon as it is free.

2.2 Graph Storage

Graphs can be stored in various formats, a suitable format is selected based on hardware and software constraints/requirements. *Adjacency Matrix* and *Adjacency list* representations are amongst the most elementary storage for-

mats. As the name suggests, *adjacency list* format is stored as an array of lists where i^{th} element of the array stores a list of neighbors of vertex i. *Adjacency Matrix* format stores the relational information in a boolean matrix format. Let the underlying graph be $G = (V, E)$ and its adjacency matrix representation be A. Then $A = |V| \times |V|$ is defined as:

$$A_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{Otherwise} \end{cases}$$

For most real world graphs, the underlying adjacency matrix is sparse. Hence, it is effective to be stored in sparse matrix formats. The computing community uses various sparse matrix formats such as *ELL-Pack* (ELL), *COOrdinate* (COO), *Compressed Sparse Column* (CSC), *Compressed Sparse Row* (CSR), *Blocked Compressed Sparse Row* (BSR), etc.

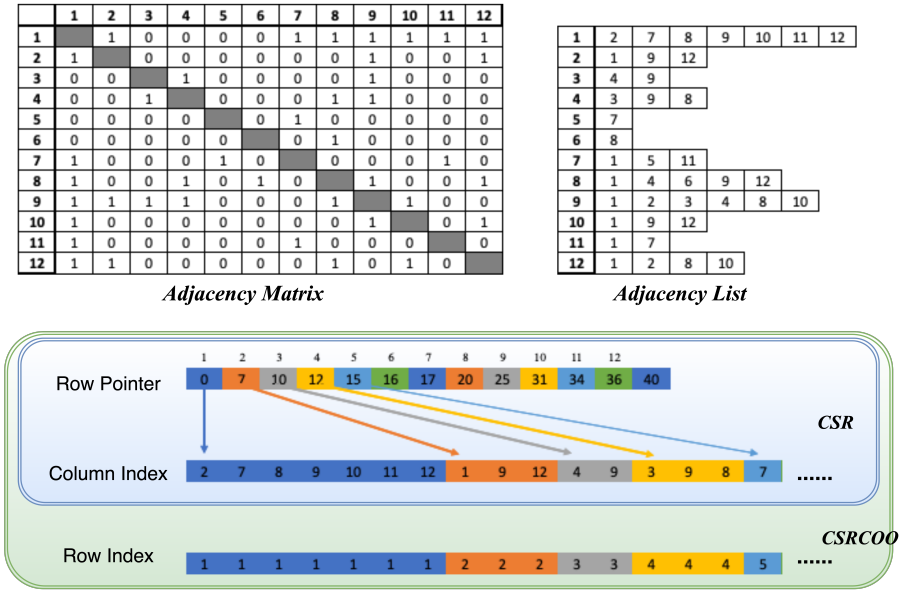


Figure 2.5: Graph Storage Formats for graph in Fig. 1.1

We use a combination of CSR and COO formats in our implementation, it is referred as the CSRCOO (Column Sparse Row with COOrdinate) format. The CSRCOO format uses 3 arrays for storing information of the underlying graph. These arrays are *Row Pointer* (length: $|V| + 1$), *Row Index* (length: $|E|$) and *Column Index:* (length: $|E|$). We maintain the *Row Index* and *Column Index* arrays sorted by their labels to enable binary search. Figure 2.5 shows comparison between different graph storage formats.

CHAPTER 3

LITERATURE REVIEW

This chapter describes relevant work in the area of subgraph enumeration in sequential (CPU) and parallel computing (multi-core/ GPU) communities. First, some definitions and notation are set to develop formal understanding.

3.1 Notation and Definitions

A graph is a pair of sets $G = (V, E)$, where V is a set of elements called vertices and E is a set of vertex pairs whose elements are called edges. An edge is represented by the elements in the vertex pair (u, v). u, v are called the endpoints of the edge. Edges of the graph can be directed or undirected. Undirected edges imply $\{(u, v) \in E \Leftrightarrow (v, u) \in E \quad \forall\ u, v \in V\}$. An edge $e = (x, y)$ is incident to a vertex v if $x = v$ or $y = v$. For the scope of this work, all graphs have undirected edges unless specifically mentioned. Graphs with directed Edge set are called directed graphs. $\mathcal{V}(G)$ and $\mathcal{E}(G)$ denote the vertex and edge set of graph G, respectively.

Neighbor set of a vertex v, denoted by $\mathcal{N}(v)$ is a set of all vertices u such that $\{u \in \mathcal{N}(v) \Leftrightarrow (u, v) \in E\}$. In case of directed graph the neighbors are further categorized as *forward/outgoing* neighbors and *backward/incoming* neighbors based on the direction of the incident edge. Degree of a vertex v, denoted by $d(v)$ or $degree(v)$ is the cardinality of its neighbor set. i.e. $d(v) = |\mathcal{N}(v)|$. Maximum degree of a graph is the maximum of all degrees from its vertex set. Oriented graph is a transformation of an undirected graph G to a directed graph $\tilde{G}$ such that:

$$\{\forall\ (u, v)\ \&\ (v, u) \in \mathcal{E}(G),\ \text{either } (u, v) \text{ or } (v, u) \in \mathcal{E}(\tilde{G})\}$$

A subgraph g of a graph G, denoted by $g \subseteq G$ is a graph $g = (V_g, E_g)$ such that $V_g \subseteq V$ and $E_g \subseteq E$ where, $\forall\ (u_g, v_g) \in E_g \Rightarrow \{u_g, v_g\} \in V_g$. Induced

subgraph by a vertex v' in graph G, denoted by $G(v')$ is a subgraph $g = (V', E')$ of G such that $V' = \mathcal{N}(v') \cup \{v'\}$ and $\{(x, y) \in E' \Leftrightarrow \{x, y\} \in V'\}$.

For the rest of this thesis, the data graph is denoted by G_d and the query graph is denoted by G_q. $V_d = \mathcal{V}(G_d), V_q = \mathcal{V}(G_q)$. As introduced, the subgraph enumeration problem is to find all instances of G_q that are isomorphic to G_d. Isomorphism is a bijection f between vertex set of two graphs, here G_q and $g \subseteq G_d$, such that $(u, v) \in \mathcal{E}(G_q) \Leftrightarrow (f(u), f(v)) \in \mathcal{E}(g)$.

As we will see in Section 3.2, the traversal based algorithms for subgraph matching consist routines for converting a given query graph to a directed graph and matching each vertex in the query graph to suitable vertices in the data graph. Whenever there are more than one matches, each of them has to be separately considered which creates branches and constructs a tree which is called the *search tree*. The order in which the data graph vertices are matched with the query graph is called *match order*. Formally, given G_q the match order π is the permutation of vertices V_q which reflects the order in which they will be matched.

3.2 Related Work

The underlying routine of subgraph enumeration is graph isomorphism. The first algorithm for detecting graph isomorphism was given by Ullman J. R. [15], here an isomorphic subgraph was found using a brute force search tree traversal with elimination of successor nodes. The traversal used here was DFS with each instance being represented as a binary table of $|V_q| \times |V_d|$ entries. For finding all instances, a DFS is recursively performed on all possible candidates with some of them pruned in advance. These search tree based algorithms are sometimes also referred as *state space search* algorithms.

Some other promising techniques which perform traversal to find query instances are exploration [16], [17], [18], [19] and constraint programming [20], [21] based. Approaches which do not perform traversal are based on converting query instances to a multi-way join framework where the edges represent relations in Gq and evaluate the multi-way join to find all results [22], [23]. However, these techniques perform better for labelled subgraph enumeration [24] and hence will be ignored in this review.

The algorithm from Ullman [15] formed a basis for various traversal based

algorithms, improvements came from different pruning heuristics and better match ordering. These algorithms are explained in Section 3.2.1. Most sequential algorithms perform state space search process at each root node. With advancements in multi-core processors and cloud computing servers, multiple scalable algorithms were developed which are mentioned in Section 3.2.2. Most recently, the focus has shifted to GPUs because of increasing memory capacity and their advent in compute servers, these contributions are mentioned in Section 3.2.3.

3.2.1 Sequential Algorithms

VF [25] reduced the memory requirements of [15] by using a State Space Representation (SSR) and developed a deterministic matching method for simultaneously verifying isomorphism. VF2 [26] introduced five feasibility rules for early pruning of the search tree to shrink the search space. VF3 [27] developed multiple additional heuristics for early pruning and further reduced the search space, these heuristics were developed based on extensive computational experimentation. Grochow *et al.* [28] assigned partial ordering to vertices matched with symmetrical query vertex to reduce redundant work. $\text{Turbo}_{\text{iso}}$ [29] improved match ordering by dynamically computing a match order for each candidate, this made the search tree leaner and provided performance improvement. $\text{Boost}_{\text{iso}}$ [30] in a novel contribution utilized vertex relationships to efficiently arrange candidate vertices to reuse adjacency list intersections.

These sequential algorithms offer great insights in the search tree exploration phase. Since, the problem has so much independent work, a sequential processor tends to be slower due to repetition of tasks. Hence, sequential implementations are only practical for data graphs with up to 100,000 vertices. Many techniques used by parallel algorithms for shrinking search space arise from sequential algorithms.

3.2.2 Multi-core Algorithms

Parallel Subgraph Listing [31] was the first truly parallel search space traversal based algorithm on a multi core hardware. It introduced a subgraph

listing framework which statically divided work across multiple threads. Various load balance strategies were developed to improve its performance, it was proved that the problem of partial subgraph distribution for workload balance is strongly NP-Hard. *CECI* [32] proposed a novel framework for Subgraph Listing based on Compact Embedding Cluster Index, this divides the data graph into multiple embedding clusters for parallel processing. Even though after orders of speedups and scalability, the technique uses BFS and needs huge amount of memory for storing intermediate results. *LIGHT* [33] eliminated the memory requirements of most parallel algorithms by developing a multi-threaded DFS framework. *Dryadic* [34] Is the current state-of-the-art CPU subgraph enumerator, it uses a robust computation tree structure which optimizes by mapping to different backend hardware to perform custom compiled graph pattern matching.

To summarize, *Multi-core* algorithms easily outperform the sequential algorithms but are limited by the parallelism offered by CPUs. Memory bandwidth across different nodes makes it further difficult to scale. BFS based strategies are prevalent in the parallel computing community which impose huge hardware requirements due to significant memory usage.

3.2.3 GPU algorithms

Due to advancements in graph algorithms on GPUs like max-flow, shortest paths and spanning trees many state space traversal based subgraph enumeration algorithms were developed. To the best of our knowledge, *GPSM* [35] is the first subgraph enumeration implementation on GPU. It used advances in BFS on GPUs to use on the basic search tree traversal algorithm. GPSM clearly outperformed the most advanced multi-core algorithm by an order of magnitude. *RPS* [36] focused on reducing the number of set intersection operations as that is the most expensive computational step by deploying a reuse framework in BFS traversal. *PBE* [37] developed a scalable framework by using graph partitioning strategies and the heterogeneous computing environment to enable multiple GPUs working on each partition.

Since, all BFS based approaches explode in memory requirements, *PARSEC* [1] developed the first DFS based subgraph enumeration on GPU. It reduced the search space by limiting compatibility to queries with at least

one fully connected node (*Central node*).

PARSEC [1] is used as a baseline for our computational experiments. It clearly showed that DFS based techniques are superior to the traditional BFS approaches on GPUs. However, it does redundant work due to unavailability of intermediate candidates and multiple processing units working on the same subtree. The parallelism schemes used by PARSEC have load imbalance that hampers its performance. We utilize the DFS framework developed by PARSEC and use different techniques from the traditional sequential and modern parallel algorithms to further improve performance.

CHAPTER 4

BASIC TECHNIQUES

The Subgraph Enumeration algorithm can be decomposed into three major steps: (1) Query Graph Preprocessing, (2) Data Graph Preprocessing, and (3) Search Tree Traversal. This chapter covers each step in detail with a running example.

4.1 Query Graph Preprocessing

This section explains all processing done on the query graph to enable it for search tree traversal. The main tasks performed on the input query are: Query sequencing and Symmetry Detection. Since the query graphs are very small (less than 10 nodes, in general) and all preprocessing tasks are polynomial time complexity, this preprocessing workload is handled by the CPU. Query preprocessing is important to shrink search tree exploration, the amount of work done at each node, and redundancy elimination.

4.1.1 Query Sequencing

The query graph input to the application is undirected. The first task is to convert this undirected graph to a directed acyclic graph (DAG). The ordering of the DAG governs the order in which these vertices are matched to the data graph. The query nodes are ordered based on their likelihood of matching with the data graph. Naturally, query nodes with lesser likelihood are prioritized over others.

The query sequencing algorithm used by VF3 [27] is utilized for this task. This algorithm uses multiple criteria to estimate the likelihood of matching and sequences the nodes in decreasing order of that likelihood. Interested readers are encouraged to read about these criteria in [27]. The pseudocode is

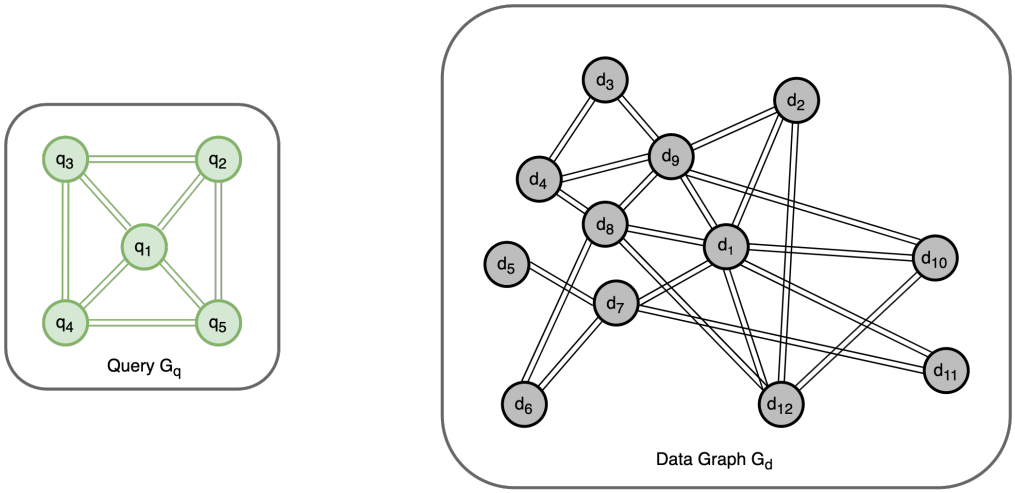


Figure 4.1: Example

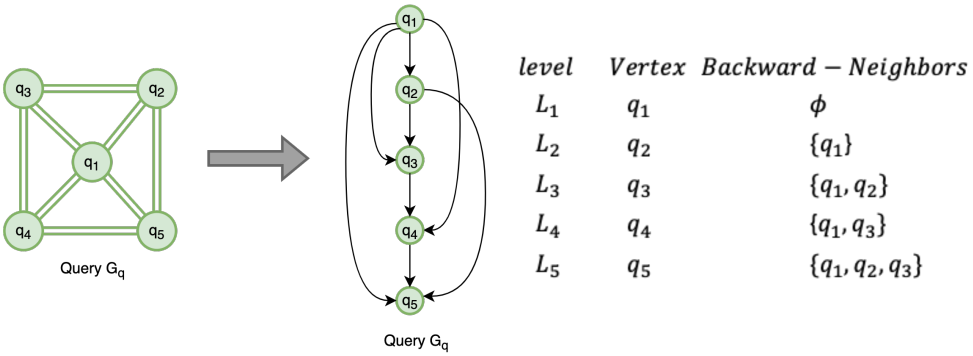


Figure 4.2: Query Sequencing output of G_q using *VF3*

listed in Algorithm 1. The order generated after performing query sequencing on the example query graph is shown in Figure 4.2.

4.1.2 Symmetry Detection and Breaking

Symmetry is the existence of automorphism in a graph. Automorphism, as the name suggests, is a graph that is isomorphic to itself. An automorphism of a graph is a permutation of vertices that maintains edges and non-edges. The set of automorphic permutations of the vertex set is called the "automorphism group". Figure 4.3 shows the automorphism group of the query graph G_q. *Orbit* is an equivalence class of vertices in the automorphism group.

The presence of symmetry essentially causes the same subgraph of the data graph to be counted multiple times and hence resulting in extra work.

Algorithm 1: Query Sequencing

Input: Template Graph, G_q
Output: Node Query Sequence, S_q

```
1  d_M[N_q] ← 0;
2  S_q ← φ;
3  for i ← 0 to N_q do
       // calculate likelihood for remaining nodes
4      for j ← 0 to N_q do
5          if j not in S_q then
6              likelihood[j] ← d_M[j] · N_q + Degree[j];
7          end
8      end
       // Find node with maximum likelihood and it to S_q
9      idx ← argmax(likelihood);
10     append idx to S_q;
       // Update d_M
11     foreach neighbor j of node idx do
12         d_M[j] ← d_M[j] + 1;
13     end
14 end
```

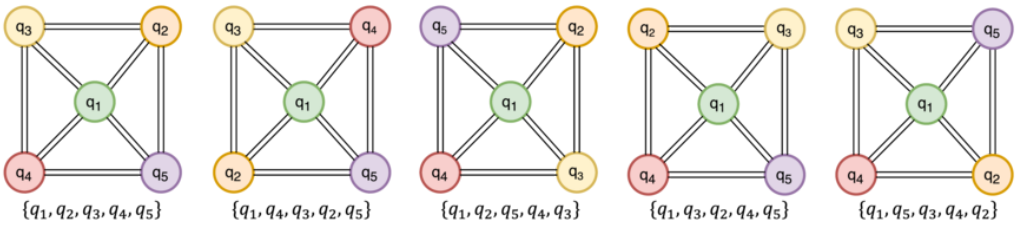


Figure 4.3: Automorphism group of G_q

The easiest technique that can be employed to avoid this is called symmetry breaking. This makes sure that the ties are resolved at the moment they are generated, hence avoiding redundant work. An easy way to break symmetry is by having an order between the matches of data graph between the symmetric levels, this was first used by [28]. Algorithm 2 lists the pseudocode for detecting symmetry and generating ordering for symmetry based pruning. Note, the `GetAutomorphismGroup` and `GetOrbits` routines used here are provided by NAUTY [38].

For example, to avoid redundancy due to automorphism $\{q_1, q_4, q_3, q_2, q_5\}$ of G_q, it can be made sure that the data graph vertices matched at level 2 have labels less than the vertices matched at level 4. This is an example

Algorithm 2: Symmetry breaking

Input: Template Graph, G_q
Output: Set of partial node orderings, $\mathcal{P}_q$

```
1 A ← GetAutomorphismGroup(G_q);
2 while A ≠ 𝕀 do
3 |   O_A ← GetOrbits(A);
4 |   Pick largest orbit {v_1, v_2, ..., v_k} from O_A;
5 |   Add v_1 < v_2, v_1 < v_3, ..., v_1 < v_k to P_q;
6 |   A ← {α|α ∈ A, α(v_1) = v_1};
7 end
```

of *lexicographic symmetry* breaking. Another symmetry breaking order can be based on the degree of data graph nodes. A key observation is that the symmetry breaking criteria can be different for different levels but not always different for different subtrees. This will be formally established and utilized in Section 5.2.

4.2 Data Graph Preprocessing

Data graph preprocessing involves operations on the data graph to shrink the search tree during traversal. With GPU implementations, preprocessing also helps to improve coalesced memory accesses and memory utilization.

4.2.1 Peeling

Subgraph Enumeration produces isomorphic matches of query graph in the data graph. Naturally, each match needs to have degree greater than the degree of the corresponding query node. This implies that: *Vertices in data graph with degree less than minimum query degree will never form any matches.* Using this simple observation, all the vertices with degree lesser than the minimum query degree can be deleted from the data graph. This can also be done iteratively as the lemma holds for the resulting data graph. This process is commonly called *Peeling*. Empirical experiments performed by [1] show that *Peeling* should be performed till at most 5% of nodes are deleted. Algorithm 3 shows the pseudocode for the peeling routine. Note the filtering of vertices is done on GPU using the CUB library [39].

Algorithm 3: Peeling data graph

Input: Minimum query graph degree, $d_{q,min}$, $G_d = (V_d, E_d)$:

repeat
  PrunedVertices $\leftarrow \{u | u \in V_d, \texttt{Degree}(u) < d_{q,min}\}$;
  if |PrunedVertices| $= 0$ **then**
    Break;
  end
  PrunedEdges $\leftarrow \{(u, v) | u$ **or** $\in$ PrunedVertices;
  $V_d \leftarrow V_d \setminus$ PrunedVertices;
  $E_d \leftarrow E_d \setminus$ PrunedEdges;
until |PrunedVertices| $< 0.05 \times |V_d|$;

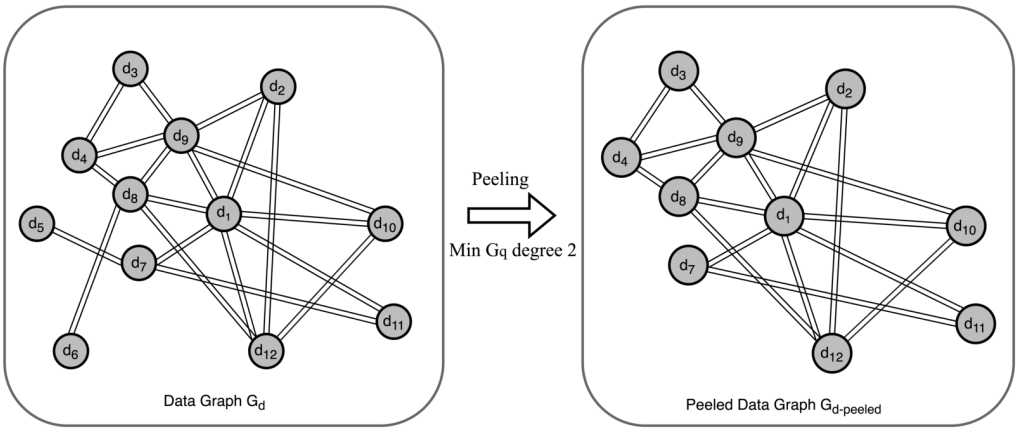


Figure 4.4: Peeling Example

4.2.2 Priority Sorting

CSRCOO storage format allows efficient storage of graphs in CPU memory. As mentioned in Section 2.2, the *Column Index* array is sorted by vertex labels to enable binary search. However, efficient symmetry breaking demands the neighbors to be listed in a different order. To achieve this, we need another copy of the *Column Index* array in a *priority* sorted fashion while maintaining the same *Row Pointer* boundaries. This array is named *Priority Sorted Column Index* or *PSCI*. The *priority* can be any property of the vertices; in our case we explore two properties: Degree and Degeneracy. Given the size of data graph we use GPUs for this preprocessing task. The *PSCI* array can be generated using a series of two stable sorts, since sorting is fast on GPUs this can be achieved by using the CUB library. To start with, the *PSCI* array is taken as a copy of the original *Column Index* array.

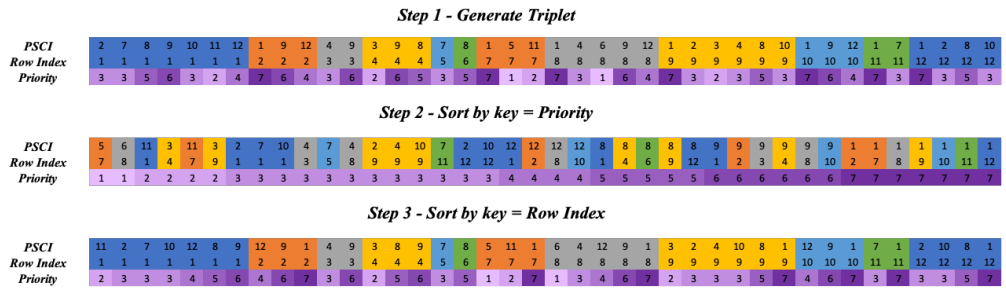


Figure 4.5: Generating Priority Sorted Column Index Array [SK: Is this image alright?]

An array of triplets is then created with each entry containing an element from *PSCI*, *Row Index*, and *priority*. Where, *priority* is an array with entries corresponding to the required property in *PSCI*. This array of triplets is subjected to two key based stable sorts, the first sort is with the keys being entries in the *priority* array while second sort is with the keys being entries in the *Row Index* array. Figure 4.5 shows the priority sorting performed on G_d as a series of key sorts.

4.2.3 Induced Subgraph

The search space in subgraph enumeration problem is essentially the whole graph for a general query. This makes the problem extremely challenging to scale. However, most practical applications of subgraph isomorphism focus on dense templates. Similar to [40], we note that restricting the search space can provide immense performance improvements, especially on the GPUs due to enhanced memory utilization.

The search space can be restricted by focusing on query graphs containing at least one *Central node*. A central node is defined as a node in the template graph that is connected to all other nodes. Under the assumption of a central node, the search space for each match is limited to the subgraph induced by the graph vertex matched to the central node. Figure 4.6 shows the subgraph induced by vertex d_{12}. To save storage space, the induced subgraph is stored in a binary adjacency matrix format. The binary format allows fast set intersection operations as well as efficient symmetry breaking [40]. To use priority based symmetry breaking, the columns of the adjacency matrix are ordered with *Priority Sorted Column Index* Array.

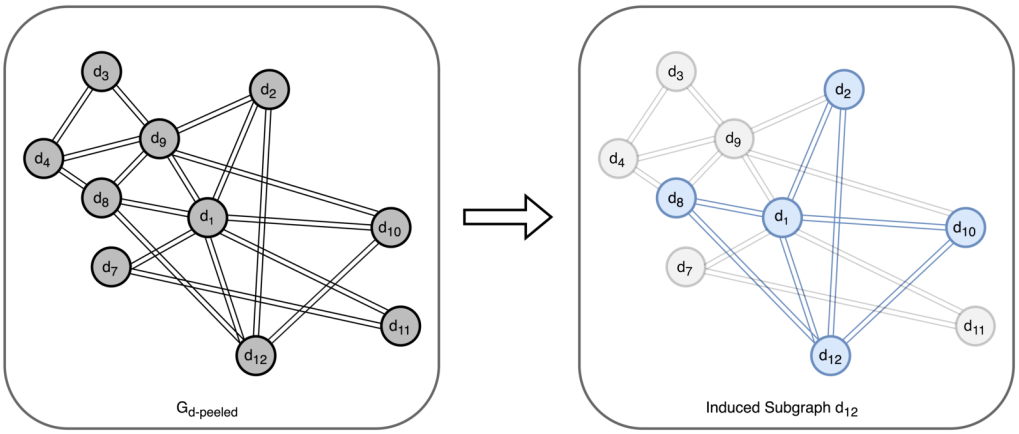


Figure 4.6: Induced Subgraph for vertex d_{12}

4.3 Search Tree Traversal

This section explains how the sequenced query graph G_q and processed data graph G_d are used to perform the search tree traversal for enumerating all matches.

Algorithm 4 describes the DFS based iterative search tree traversal process. The whole traversal can work within a DFS stack of size $|V_q| \times Degree_{max}(G_d)$. At each node of the search tree, *GetNextNodes* function is called which generates the candidates to be visited in the next level. This function is described in Algorithm 5. It performs an adjacency list intersection operation to generate all possible candidates for the next level (lines 2 - 5). Since, there might be redundant candidates generated due to the symmetry of the query, some of them need to be pruned. This is done by the next loop (lines 6 - 8). Note that the set $\mathcal{V}_{orient}$ is available without overheads due to column order of the induced subgraph.

We now walk through the Search tree traversal algorithm with the example query graph sequence given in Figure 4.2 and the peeled data graph in Figure 4.4. Note, the pseudocode in Algorithm 4 describes a DFS-based traversal technique while the example enumerates search tree traversal in a BFS manner for illustration purposes.

Step 1: Select candidates in G_d with degree greater than or equal to q_1. Here only vertices $\{d_1, d_8, d_9, d_{12}\}$ will be selected. All these are matched to q_1 and traversed separately. For conciseness, we will only consider vertices d_1 and d_{12}. These matches are shown in Figure 4.7.

Algorithm 4: DFS Traversal

Input: Sequenced Query Graph: $G_q = (V_q, E_q), S_q$
Induced Subgraph: G^d_{ind}. Initial level: l_{init}
Initial set of Nodes at l_{init}: $\texttt{matches}_{init}$ and size $\texttt{num\_matches}_{init}$

Output: Set of enumerated matches and count: `final_enum`, `count`

```
1  l ← l_init, matches[l] ←matches_init, num_matches[l] ←num_matches_init;
2  curr_idx ← 0, final_enum ← φ, count ← 0 ;
3  while curr_idx[l] ≤ num_matches[l] do
4      GenNextCandidates ();
5      num_matches[l + 1] = |matches[l + 1]|;
6      if (num_matches > 0 and l < |V_q| − 1) then
7          l ← l + 1;
8          curr_idx[l] ← 0;
9      end
10     else if l == |V_q| − 1 then
11         for n = l_init to num_matches[l + 1] do
12             match ← φ;
13             for k = l_init to l do
14                 match ← match ∪ matches[k][curr_idx[k]]
15             end
16             final_enum ← final_enum ∪ match
17         end
18         count ← count + num_matches[l + 1]
19     end
20     else
21         curr_idx[l] ← curr_idx[l] + 1;
22         while curr_idx[l] == num_matches[l] and l > l_init do
23             l ← l − 1;
24             currIdx[l] ← curr_idx[l] + 1;
25         end
26     end
27 end
```

Figure 4.7: Step 1

Algorithm 5: Generate Candidates for Next Level

Input: Induced Subgraph G^d_{ind}, level l
Backward Neighbor List $\mathcal{N}'$
Symmetry levels List $\mathcal{S}$
Set of vertices of lower priority than given vertex $\mathcal{V}_{orient}()$
List of matches in previous levels `matches[]`

```
1  Function GenNextCandidates()
   while (curr_idx[l] < num_matches[l]) do
2      forall k ∈ N'(S_q[l+1]) do
3          u ← matches[k][curr_idx[k]];
4          matches[l+1] ← matches[l+1] ∩ N(G^d_ind(u));
5      end
6      forall j ∈ S(S_q[l+1]) do
7          u ← matches[j];
8          matches[l+1] ← matches[l+1] \ V_orient(u);
9      end
10 end
11 Function End
```

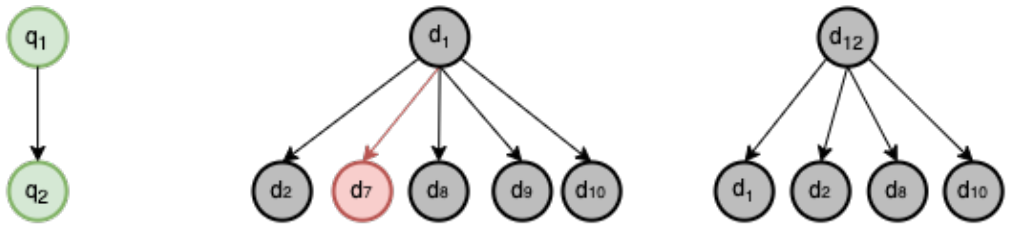


Figure 4.8: Step 2

Step 2: For candidates of level 2, select all neighbors of vertex matched at level 1 with degree greater than that of q_2, i.e., 3. As shown in Figure 4.8 for the subtree rooted at d_1, there will be 5 candidates: $\{d_2, d_7, d_8, d_9, d_{10}, d_{12}\}$. Out of these 5 candidates, d_7 is pruned since, it has degree less than q_2.

For subtree rooted at d_{12}, there will be 4 candidates: $\{d_1, d_2, d_8, d_{10}\}$ and none can be pruned.

Step 3: Level 3 candidates are obtained by the intersection of matches at level 2 and level 1. Since this level is symmetric to level 2 symmetry breaking needs to be performed. We use the decreasing degree criteria for symmetry breaking in this example. Figure 4.9

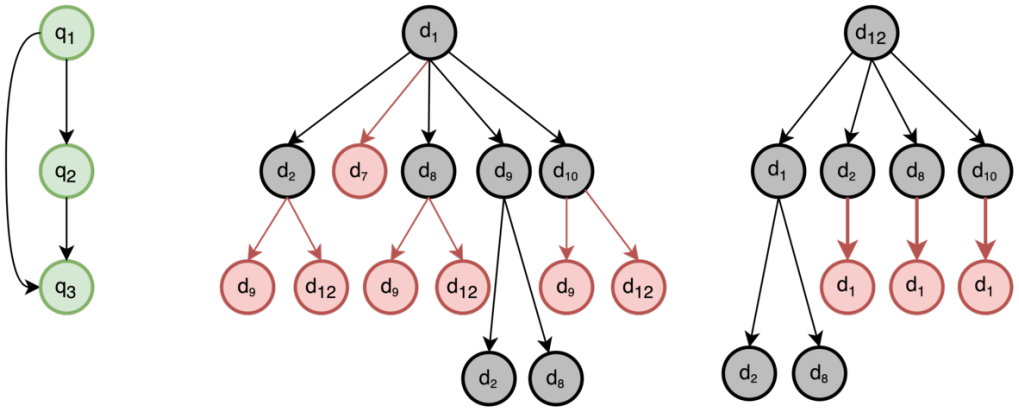


Figure 4.9: Step 3

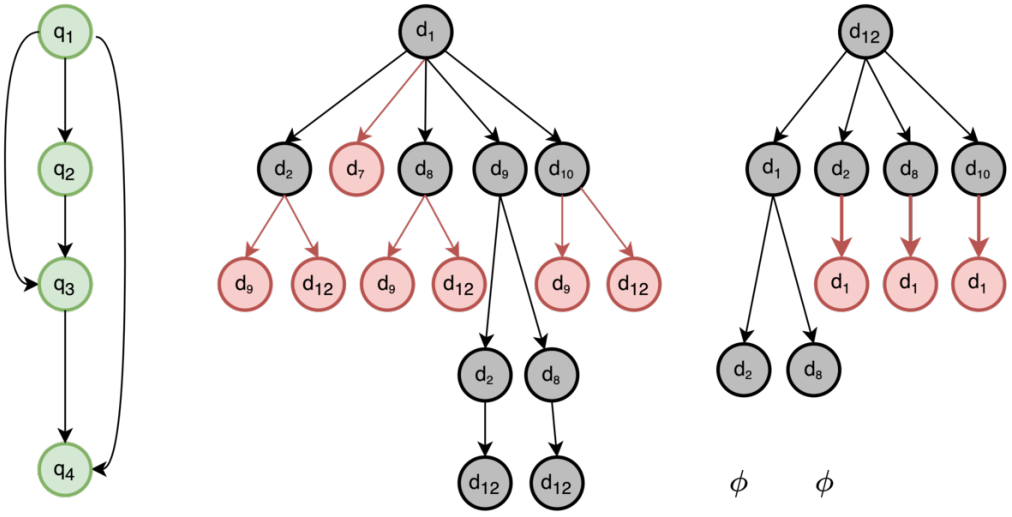


Figure 4.10: Step 4

shows all candidates at level 3. The candidates marked in red are pruned. For example 9 and 12 is pruned in subtree $d_1, d_8, \ldots$ since $Degree(d_9) < Degree(d_8)$.

Step 4: For level 4 candidates, the adjacency intersection of q_1 and q_3 is required. This level is also symmetric to level 2. The subtree rooted at d_{12} does not yield any candidates. While the other gets one candidate each. Symmetry Breaking does not prune any candidates here.

Step 5: Level 5 candidates (or final matches) are generated by the adjacency intersection q_1, q_2, and q_4. This level is symmetric to both q_2 and

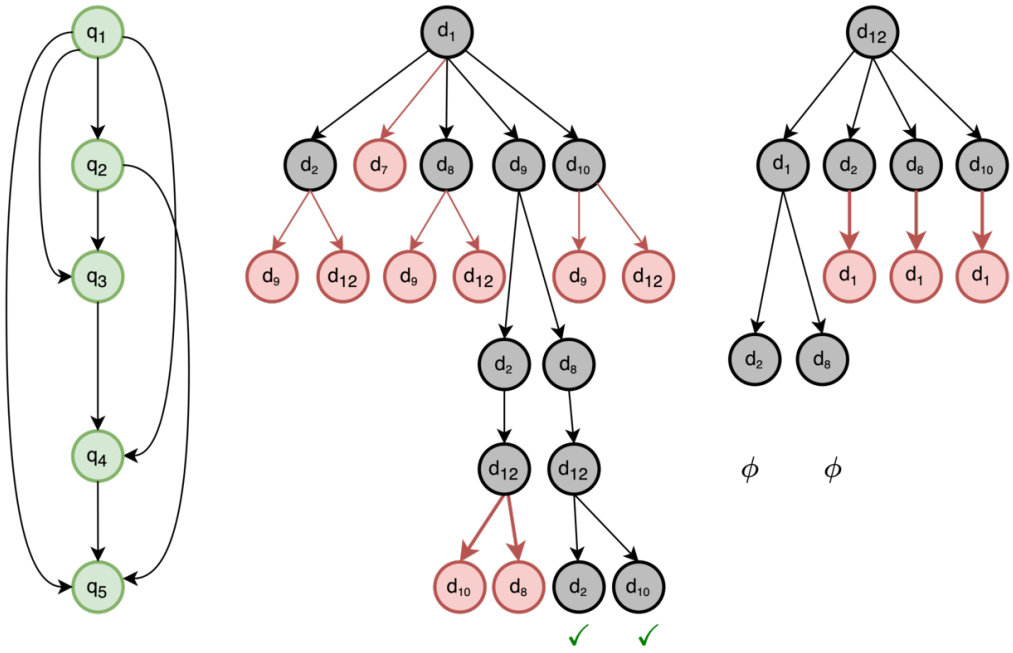


Figure 4.11: Step 5

q_3. The candidates generated by intersection at the first leg are $\{d_8, d_{10}\}$, symmetry breaking will eliminate both the candidates as $Degree(d_2) < Degree(d_8)$. [RN: DONE TILL HERE.] There is a tie between d_2 and d_{10} which is resolved *lexicographically*. Similarly, the other leg generates 2 valid candidates for level 5.

This way the search tree traversal process terminates with final matches $\{d_1, d_9, d_8, d_{12}, d_2\}$ and $\{d_1, d_9, d_8, d_{12}, d_{10}\}$. There might be more matches originating from subtrees rooted at $\{d_8, d_9\}$ which were not illustrated here.

To conclude, this chapter gives a detailed understanding of all techniques involved in subgraph enumeration. As we can see, all subtrees are independent of each other hence, they can be easily parallelized on GPUs.

One more subtle observation to be made here is that the exploration is dependent on the symmetry breaking technique. We can see from Figure 4.9 that there would have been many more candidates at level 4 if the symmetry breaking was done using lexicographic criteria or increasing degree criteria. These candidates would have ultimately pruned in lower levels, resulting in extra work. Therefore, selecting an efficient symmetry-breaking strategy is important and discussed in detail later in Section 5.2.

CHAPTER 5

IMPROVEMENTS

Multiple ideas were applied to improve the performance of baseline, PARSEC [1]. This chapter lists each idea and its individual impact on performance. All improvements are then combined to present final results in Chapter 6. The basic philosophy used is to find the most time-consuming step and try to make it faster using algorithmic or computational techniques.

The criteria used for quantifying the contributions of these improvements are number of intersections and runtime. These numbers are reported as speedups, the ratio of old to new. Formally, we define the intersection speedup and time speedup as:

$$\text{Intersection Speedup} = \frac{\text{\# of Intersections Baseline}}{\text{\# of Intersections New}}$$

$$\text{Time Speedup} = \frac{\text{Run time Baseline}}{\text{Run time New}}$$

The information of data graphs and queries used to report individual performance impact is mentioned in Section 6.1.

5.1 Intersection Reuse

Set intersection operation for generating candidates at the next level is the most time-consuming operation in subgraph enumeration, this is a well-established fact in the literature [36], [33], [27]. This was also verified for the baseline [1]. Figure 5.1 gives the distribution of time spent by each operation in different tasks. The Intersect operation shown here is the time taken by Algorithm 5. This algorithm involves iterating over each element of the backward element list to perform intersection and generate possible candidates for next level (lines 2 - 5). The number of backward neighbors at

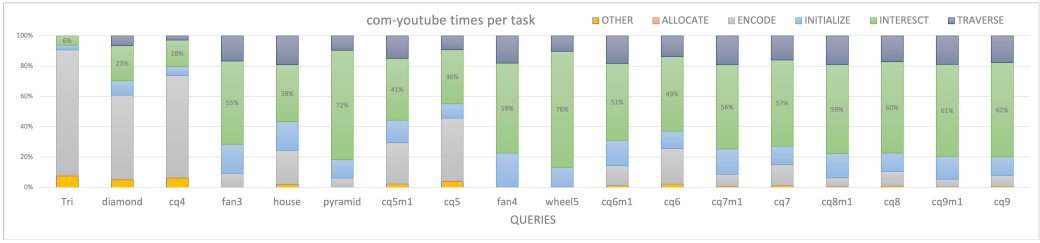


Figure 5.1: Stepwise Fraction Time spent

each level increases with increasing template size. This results in even more adjacency list intersections at each node. RPS [36] reduces these operations by generating an intersection reuse plan, this plan smartly finds the intersections that will be required by more than one node and stores them when first calculated. There is a lot of intersection reuse possible with RPS as they employ a BFS strategy. Similar extent of reuse is not possible in DFS traversal since the past information is lost while backtracking. However, Some levels of the sequenced query graphs have similar backward neighbor lists. For such queries, the majority of intersections can be reused by simply storing the intermediate intersection results. To do this efficiently, one needs to match levels with similar adjacency lists. This problem is posed as a linear programming optimization problem.

Let, the sequenced query graph be $G_q = (V_q, E_q)$ with $|V_q| = k$ and the vertex sequence S_q. $\mathcal{N}(.)$ be the function for getting the backward neighbor list of a vertex. For each pair of vertices at level i and j let W_{ij} be a measure of the commonality between their backward neighbors'. Let $X_{i,j}$ be the decision variable which tells if vertex i should reuse intersections from vertex j.

With these definitions the optimization problem can be modelled as:

$$\max \sum_{i=j+1}^{k} \sum_{j=1}^{k} W_{ij} X_{ij} \tag{5.1}$$

$$\text{s.t.} \sum_{j=1}^{k} X_{ij} \leq 1. \quad \forall i \in \{1, \ldots, k\} \tag{5.2}$$

Where,

$$W_{ij} = \begin{cases} |\mathcal{N}(S_q[i]) \cap \mathcal{N}(S_q[j])| & \text{if} \quad i > j, \mathcal{N}(S_q[i]) \supseteq \mathcal{N}(S_q[j]) \\ 0 \qquad \text{Otherwise} \end{cases}$$

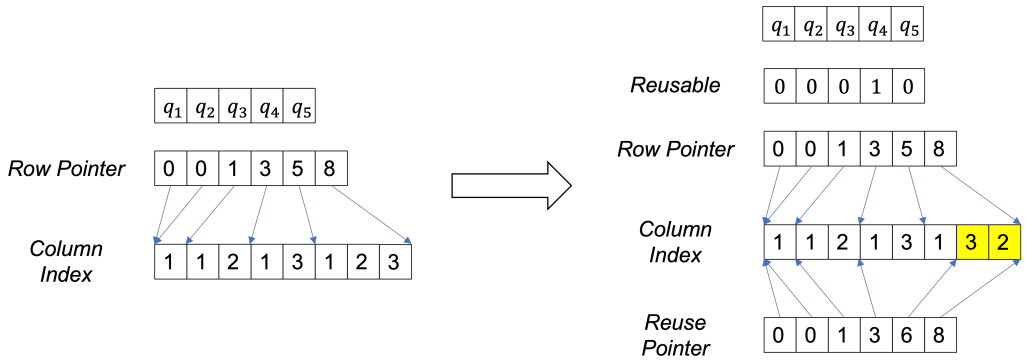


Figure 5.2: Reuse Detection Example

This problem is a linear semi-assignment problem (LSAP) where the greedy solution is optimal. The solution to this problem is:

$$X_{ij} = \begin{cases} 1 & \text{if } j = \texttt{argmax}_j(w_{ij} > 0); \\ 0 & \text{Otherwise} \end{cases}$$

Reuse detection involves a find minimum operation for each level in G_q hence it is polynomial time and can be performed on CPU for small-sized query graphs. The mappings found $x_{ij} = 1$ implies that level i can use intersection results from level j. Hence, the level j is flagged as *reusable* and its results are stored along with the DFS stack. Algorithm 6 lists the reuse detection and post-processing recipe. To save on constant memory the query graphs are stored in CSR format. After finding the reuse mappings from the query, the column index of (G_q) needs to be modified for the search tree traversal step to utilize this information. The modification step involves reordering the lists and generating the reuse pointer array.

We show reuse detection at work on the example query in Figure 4.1. Here, only w_{54} and w_{53} are positive with value 2. Hence, any of the levels 3 or 4 can be marked reusable, Let's assume level 4. The backward neighbors of level four are $\{1, 3\}$ and for level five they are $\{1, 2, 3\}$. To avoid checks in the for loop of Algorithm 5 (line 2). Another Reuse pointer array is created which gives the start of the remaining neighbors of level 5.

Once the reuse is detected and pointers set, this data is sent to the GPU constant memory for the rest of the application. For implementing reuse, the function in Algorithm 5 needs to be changed to add conditions for checking if a level is reusable or if the reuse pointer is updated. This is shown as a flow

chart in Figure 5.3. This way, reuse reduces computational workload and adds to the memory workload. Since global memory writes are relatively expensive, some shared memory can be utilized to write the intermediate intersection results while longer results can be spilled to global memory.

Algorithm 6: Reuse Detection

Input: Directed Query Graph $G_q = (V_q, E_q)$
Arrray of Backward Neighbor lists $\mathcal{N}(V_q)$

```
1  W ←: [|V_q|][|V_q|];
2  X ←: [|V_q|][|V_q|];
3  reusable ←: [|V_q|];
4  forall j ∈ V_q do
5  |  for i = j + 1 to |V_q| do
6  |  |  if N(V_q)[i] ⊇ N(V_q)[j] then W[i][j] ← |N(V_q)[i] ∩ N(V_q)[j]|;
7  |  |  else W[i][j] ← 0 ;
8  |  end
9  end
10 reusable[j] ← (max(W[i].) > 0);
11 forall i ∈ V_q do
12 |  for j = 1 to i do
13 |  |  if j == argmax_j(W[i]. > 0) then 1;
14 |  |  reusable[j] = 1;
15 |  |  else 0;
16 |  end
17 end
18 // Update Reuse Pointers from X[i][j]
```

The improvements due to intersection reuse are measured in terms of the number of intersections performed. Since there are increased reads and writes to global memory the time benefit from this technique is negligible. Table 5.1 shows the improvement in the number of intersections with this technique. Since the backward neighbor lists are small for sparse queries there is not much benefit due to intersection reuse. The intersection counts also include the additional read and write operations needed due to intersection reuse.

5.2 Hybrid Symmetry Breaking

Symmetry breaking is an important operation in the search tree traversal. [15] gives a simple ordering technique for symmetry breaking, but the cri-

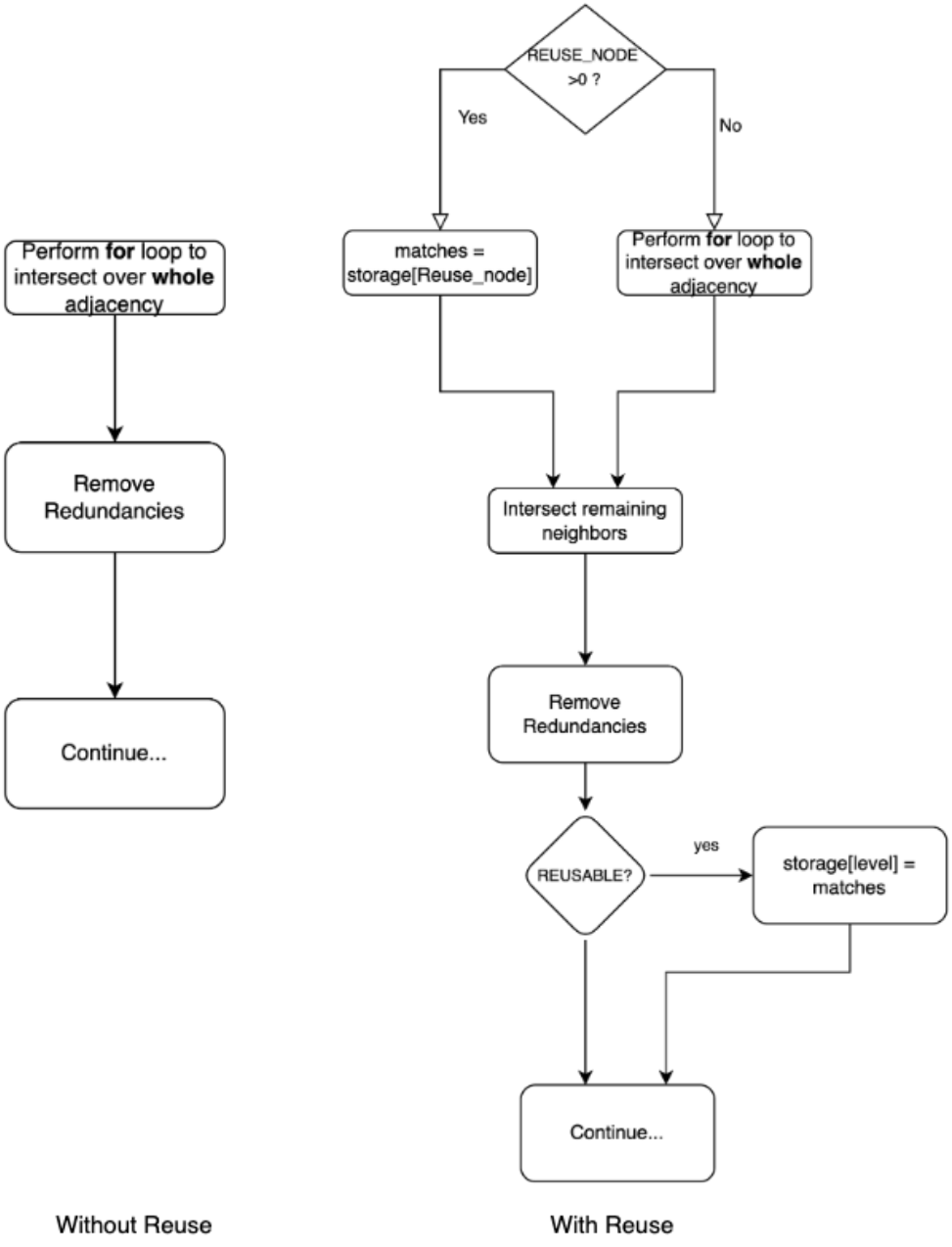


Figure 5.3: Flowchart with and without reuse

Query	#Intersections w/o Reuse	#Intersections with Reuse	Intersection Speedup
diamond	9,722,143	9,722,143	1.00
cq4	6,028,832	6,028,832	1.00
cq5_m1	46,794,486	26,360,303	1.78
cq5	20,881,838	15,889,824	1.31
cq6_m1	130,114,516	52,225,436	2.49
cq6	49,618,729	30,194,042	1.64
cq7_m1	253,215,589	81,007,445	3.13
cq7	91,679,280	46,935,236	1.95
cq8_m1	370,669,539	102,639,103	3.61
cq8	139,279,024	62,709,660	2.22
cq9_m1	430,334,955	111,122,133	3.87
cq9	181,479,780	74,663,152	2.43

Table 5.1: Number of intersections with and without reuse for com-youtube

teria for this ordering is not well-defined. It is okay to use any criteria for correctness, but good criteria can significantly improve performance. Handling symmetries to reduce the search space is an active area of research. The problem of finding an efficient symmetry breaking technique was shown to be strongly NP hard by [41]. A lot of symmetry breaking criteria have been defined since then. Most of these have linear time complexity or work only with BFS traversal. In the optimization domain, this problem has also been formulated as a Constraint Satisfaction Problem (CSP) but it also works only when the tree traversal is done in BFS fashion [42]. The near-linear time complexity for each decision, constraints to perform BFS, and data sharing between subtrees make efficient symmetry breaking difficult on GPUs with DFS strategy. Since the knowledge of *global* characteristics is limited during enumeration, we resort to static characteristics like Degree and Degeneracy. The priority sorted column index array (PSCI) (discussed in Section 4.2.2) is used to further reduce global memory accesses.

With PSCI, different symmetry breaking criteria were tried and compared to the baseline [1]. The baseline criteria are based on degree for level 1 and lexicographic for level 2 onwards.

The symmetry breaking performance is determined by finding the total number of intersections. On performing these experiments, the Degeneracy based symmetry breaking seemed to perform worse than baseline across all graphs. For degree, increasing degree based criteria seemed to perform better

for some queries while decreasing criteria performed better for others. Table 5.2 shows the Intersection speedups for the *soc-pokec* and *cit-patents* data graphs.

soc-pokec	Tri	Diamond	Cq4	Cq5m1	Cq5	House	Pyramid	Fan3	Cq6m1	Cq6
Decreasing	1.22	0.53	0.66	0.89	1.19	1.49	1.88	1.76	1.06	1.41
Increasing	3.00	1.08	1.47	1.67	2.29	2.85	0.22	0.20	1.64	2.25
cit-patents	Tri	Diamond	cq4	cq5m1	cq5	house	pyramid	fan3	cq6m1	cq6
Decreasing	1.22	0.82	1.24	2.32	2.91	4.29	3.15	3.92	2.51	3.15
Increasing	3.00	1.27	1.79	3.50	4.28	6.60	1.26	1.83	2.72	3.71

Table 5.2: Intersection Speedup with Degree based Symmetry Breaking

Recall from 4.1.2 that the symmetry breaking criteria can be different for different levels and maybe different for different subtrees. To precisely find out when it can be different and when it cannot, we present the following observation:

Theorem 1. *Symmetry breaking is independent across subtrees for central node queries with first symmetric level greater than two*

Proof. Let, the data graph be $G = (V, E)$ and $\{v_1, v_2\} \in V$. $G_{ind}(v)$ be subgraph induced by v in G. The backward neighbors list of a vertex is represented by $\mathcal{N}'(.)$.

The proof is obvious if $\mathcal{E}(G_{ind}(v_1)) \cap \mathcal{E}(G_{ind}(v_2)) = \phi$

If not $\forall\ e(x, y) \in \mathcal{E}(G_{ind}(v_1)) \cap \mathcal{E}(G_{ind}(v_2))$:

Since, $(x, y) \in \mathcal{E}(G_{ind}(v_1)) \ \Rightarrow\ \{x, y\} \in \mathcal{N}(v_1)$

Similarly, $\{x, y\} \in \mathcal{N}(v_2)$

Now for Algorithm 4 since level 2 vertex (say q_2) in the query graph is not symmetric to the level 1 vertex (say q_1), candidates for level 2 will be given by $\mathcal{N}'(q_2)$ where q_2 is the query vertex at level 2.

Since, q_1 has to be a central node, $\mathcal{N}'(q_2) = q_1 \Rightarrow$ Level 2 candidates for tree rooted at v_1 will be $\mathcal{N}(v_1)$ and for tree rooted at v_2 they will be $\mathcal{N}(v_2)$.

This implies, the level 2 candidates are independent of the symmetry breaking strategy, the edge (x, y) is always considered for exploration by Algorithm 4.

Since, all edges in the intersection are considered independent of the symmetry breaking strategy. Each subtree will enumerate the correct number of query instances. □

Theorem 1 enables different symmetry breaking strategies for different subtrees. Table 5.2 shows which strategy to choose really depends on the data graph and query graph. The optimal strategy may even be different for distinct subtrees in the same data graph. To curb this variability in strategy decisions, a dynamic two phase traversal technique is designed: The first phase is to determine if a particular subtree should perform increasing or decreasing symmetry breaking. The second phase is to use the predetermined strategy to perform the search tree traversal. Performing increasing or decreasing symmetry breaking differs by one step hence the second phase can simply check the predetermined strategy and prune accordingly.

For phase 1, the problem of finding an optimal strategy is strongly NP hard. Thus, any practical implementation can only develop Heuristics. A natural strategy to decide the strategy in phase 1 might seem to be based on counting the number of eligible candidates till first symmetric level. This does not work as both strategies will produce same number of candidates. This hints at designing a criterion to determine the *quality* of these candidates. Since the candidates generated are sorted by priority, we use the following observations to determine their *quality*:

Obs1: For increasing symmetry breaking, the left side of the DFS subtree has lesser candidates (vice versa for decreasing).

Obs2: Few candidates on the left side may generate more nodes in subsequent levels, while the right side candidates (high count) might not.

So the weights should be selected based on an equalizing philosophy where higher weight is given to the left side of the tree (for increasing) and less to the right side. Candidate wise increasing weights based on index of the candidate and the total number of candidates is chosen for this purpose.

To formalize, Let the total number of candidates at j^{th} leaf in the subtree be C_j and the total number of enumerations found till this level be S. The symmetry breaking strategy is determined based on the weighted sum (say WS).

$$WS = \sum_j C_j \times W_j$$

Query	Increasing	Decreasing	Hybrid	Best	Worst
Tri	3.00	2.20	3.00	3.00	2.20
Diamond	1.27	0.82	1.27	1.27	0.82
cq4	1.79	1.24	1.79	1.79	1.24
cq5m1	3.50	2.32	3.50	3.50	2.32
cq5	4.28	2.91	4.28	4.28	2.91
house	6.60	4.29	6.60	6.60	4.29
pyramid	1.26	2.45	2.47	2.53	1.24
fan3	1.83	3.92	3.96	4.07	1.80
cq6m1	2.72	2.51	2.72	2.72	2.51
wheel5	1.00	1.57	1.57	1.57	0.99
fan4	1.56	2.20	2.22	2.22	1.56
cq6	3.71	3.15	3.71	3.71	3.15

Table 5.3: Comparison of Hybrid strategy with pure, best and worst possible strategies using intersection speedup criteria for data graph cit-patents

The weightage W_j for the quality of this branch is given by:

$$W_j = \begin{cases} (S - j) & \text{for increasing} \\ j & \text{for decreasing;} \end{cases}$$

The strategy is chosen for subtrees based on whichever weighted sum (WS) is smaller.

To find the efficiency of this strategy, we compare it with the best, worst, and pure strategies using intersection speedup criteria. Table 5.3 shows that hybrid strategy often works better than pure strategy and is sometimes close to the best possible strategy. This improvement in reduced work reflects in time improvements. Table 5.4 gives the time speedups for *fan3* and *pyramid* across different data graphs. Note, these queries were specifically selected since there first symmetric level is greater than two. So the hybrid symmetry breaking does not rely on pure strategy. The reasons for intersection speedups not proportionally reflecting in time speedups are - Time taken by other operations, Load imbalance and 2 phase strategy overheads.

Data Graph	Queries					
	Fan3			Pyramid		
	Time (s)		Speedup	Time (s)		Speedup
	Baseline	Degree hybrid		Baseline	Degree hybrid	
soc-pokec	2.464	1.500	**1.643**	3.086	1.964	**1.571**
com-youtube	9.675	6.206	**1.559**	19.253	10.335	**1.863**
cit-patents	0.534	0.425	**1.256**	0.598	0.472	**1.265**
com-orkut	447.532	342.183	**1.308**	996.422	582.864	**1.710**
as-skitter	60.851	40.456	**1.504**	159.289	84.530	**1.884**

Table 5.4: Time speedups with hybrid Symmetry Breaking

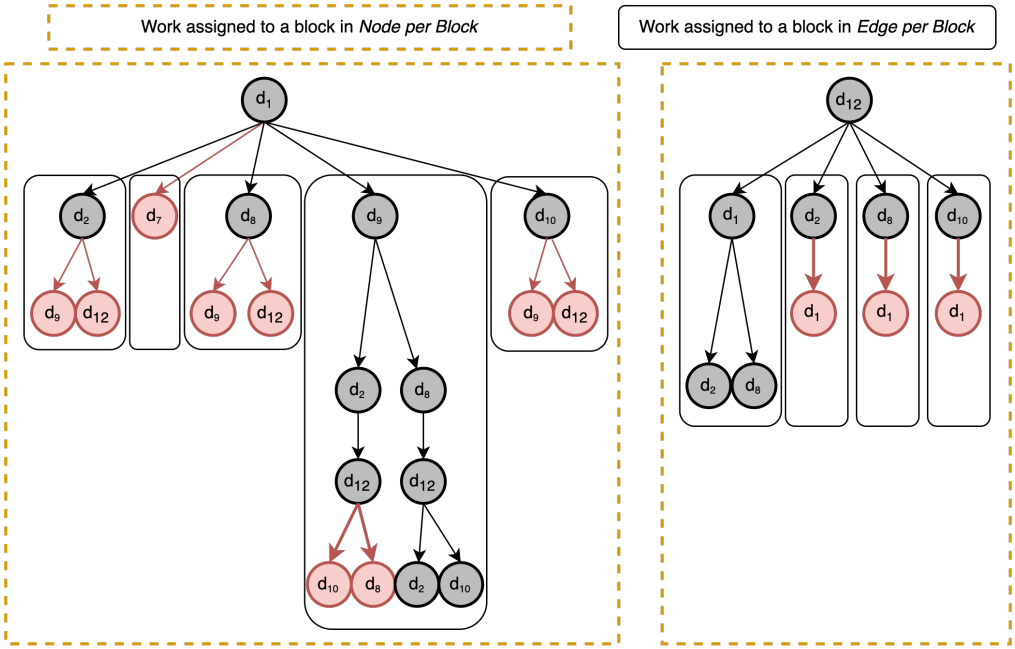


Figure 5.4: Parallelization Schemes

5.3 Hybrid Parallelism for load balance

As mentioned in Section 2.1.5 load balance is an important criterion while designing parallelization schemes. The baseline implementation [1] offered 2 types of parallelism *Node per Block* and *Edge per Block*. Figure 5.4 shows the difference between the two parallelization schemes.

Two techniques were implemented to reduce the load imbalance, (1) Task scheduling, and (2) Hybrid Parallelism. Some runtime insights from the baseline are presented to explain these improvements, these insights also quantify the observations from baseline.

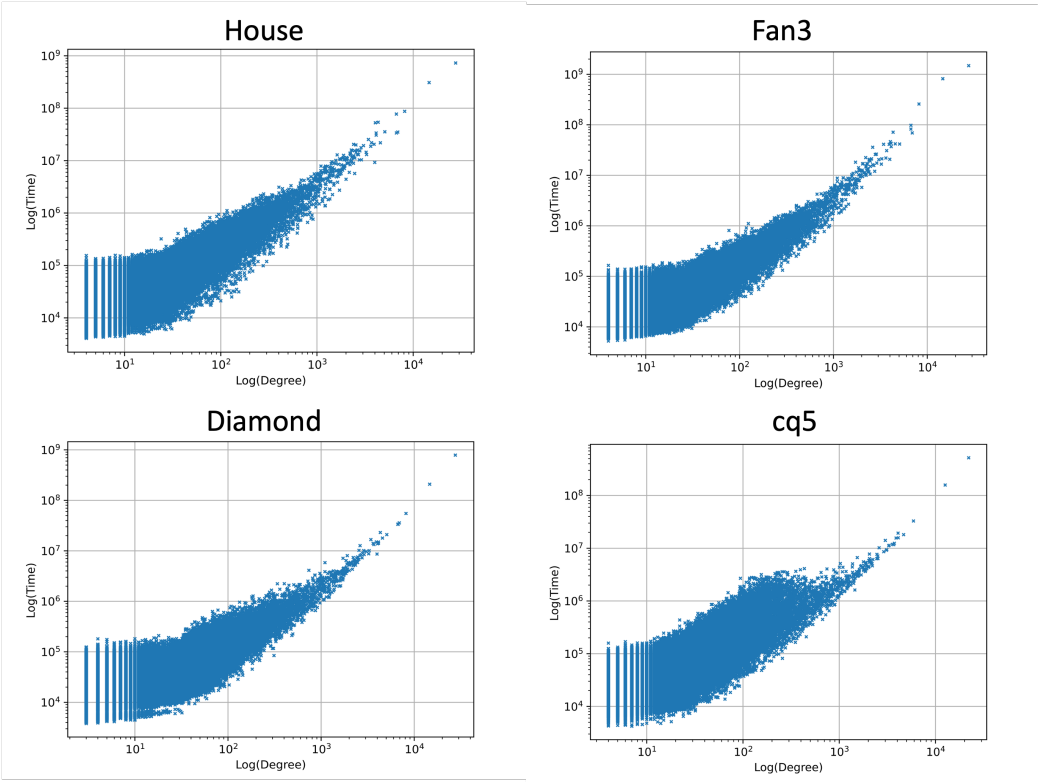


Figure 5.5: Degree vs Runtime for data graph com-youtube

Observation 1 - Run time vs Degree

Figure 5.5 shows the relationship between degree of the root node and runtime as a log-log plot. It is clear that the run time is exponentially related to the root node degree also there is huge variation in the run time at same degree.

Observation 2 - Load balance comparison

Given the trend between runtime and degree of the root node, the node per block scheme is expected to show a huge load imbalance. The load balance is supposed to improve for edge per block scheme as it would split the work offered by a high degree root node into several blocks. Figure 5.6 shows the load balance across SMs for various query graphs as standard box plot. This is computed by finding the processing time spent by each SM during the kernel lifetime. These times are then normalized to remove variability due to query graphs. It is clear from the figure that the edge per block

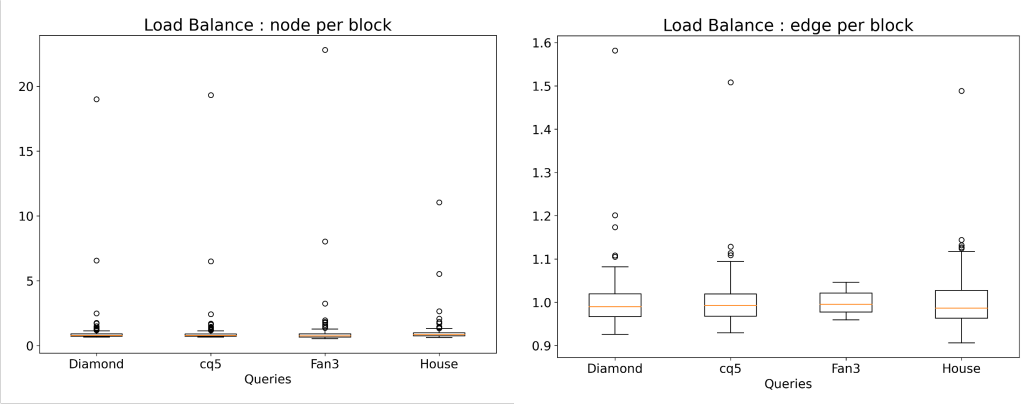


Figure 5.6: com-youtube load balance with different parallelization schemes

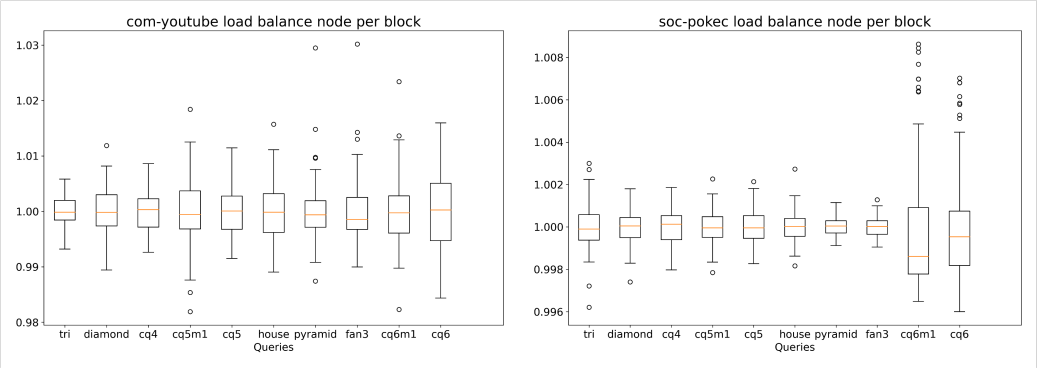


Figure 5.7: Load balance for nodes with degree ≤ 256

parallelization scheme significantly improves load balance.

Observation 3 - Load balance for low degree nodes

Since the trend between degree and runtime is exponential, the subtrees originating from nodes with low degrees will likely have lesser variation in run times. Figure 5.7 shows there is negligible load imbalance for low degree nodes. For these findings, all nodes with degree higher than cutoff were filtered out before kernel launch.

The claim from the authors of [1] was edge per block scheme works better than node per block for queries of size greater than five. One reason for this claim is implied by the load balance observations above, however, it does not explain the slow performance of the edge per block kernel. The reason here is strictly based on the implementation. For the edge per block scheme in [1], each block induces its subgraph. This causes extra workload

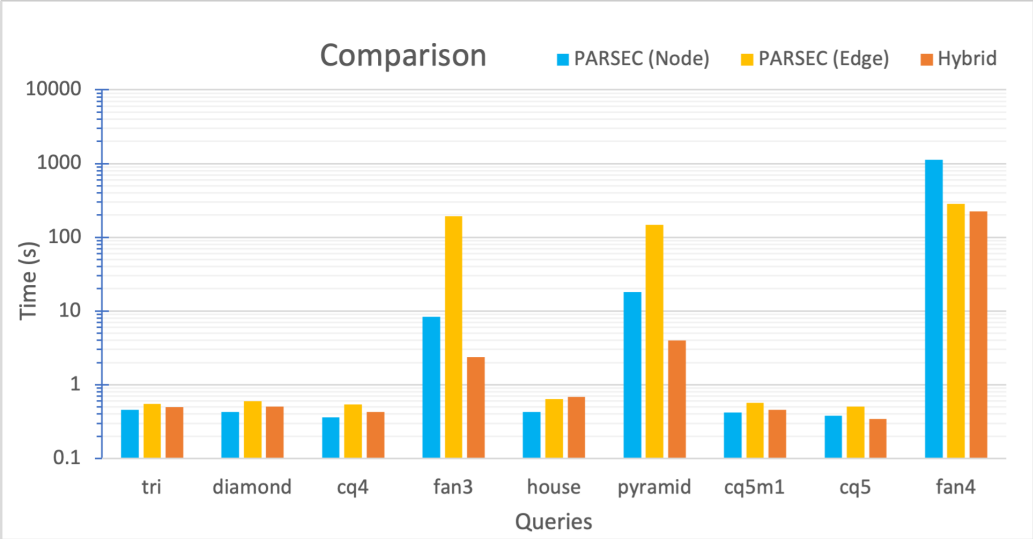


Figure 5.8: Run times with Hybrid parallelism for com-youtube

hence the edge per block implementation outperforms only for larger query graphs as they exhibit worse load balance. To remove this redundant work, the kernel is split with different parallelization schemes in each part. The first kernel ensures non-redundancy while inducing subgraphs by using the node per block scheme. The second kernel uses these induced subgraphs to perform search tree traversal using the edge per block scheme while ensuring better load balance.

Since the kernels are split, the persistent memory technique used in the baseline [1] needs to be removed. To minimize the total number of kernel launches and better resource utilization a runtime query to the device memory is performed and the number of nodes to process at once is found accordingly.

It is clear from Figure 5.7 that the low degree nodes do not exhibit load imbalance with node per block parallelization. Since, splitting the kernels loses cache benefits and causes launch overheads a cutoff scheme was developed. Nodes with degree less than cutoff would perform subgraph inducing and search tree traversal in node per block fashion while nodes with degree higher than cutoff would perform these operations as mentioned above. Figure 5.8 shows the improvement in runtimes for the com-youtube data graph across different queries with cutoff being 2048.

Empirical testing was performed to fix a cutoff value across different queries. It was found that for dense templates like *cqx* and *cqxm1* cutoff value be-

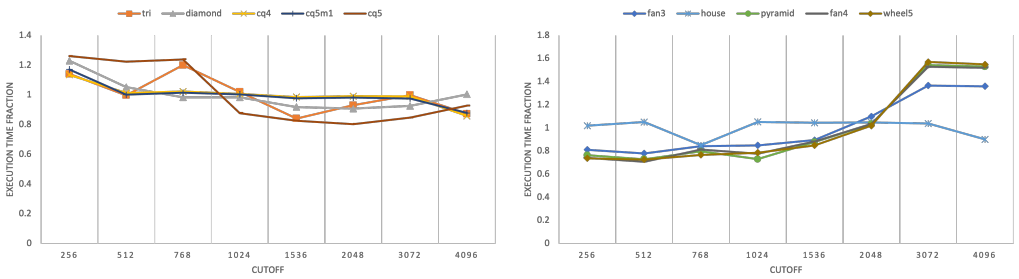


Figure 5.9: Run time fractions vs cutoff for com-youtube

tween $(1024 - 2048)$ is better while for sparse templates like *fans, pyramids,* and *wheels* cutoff value around 768 achieved best results. Figure 5.9 shows the time fraction of runtimes across different cutoffs, time fraction is defined as runtime of an instance divided by average time across all instances (Lower is better for these Figure). Using these insights the cutoff was decided at runtime based on input query.

CHAPTER 6

RESULTS

6.1 Experimental Setup

All results presented in this chapter are generated on a supercomputing hardware cluster with a dual-socket 64-core AMD EPYC 7763 ("Milan") CPU and NVIDIA A100 40GB HBM2 GPU. The system has 256 GB of DDR4 memory with 1.5 TB swap memory via high-performance-SSD. The program is compiled with NVCC (CUDA 11.7, driver version 515.48) SM version 80, and GCC 11.2 with the -O3 flag.

For a fair comparison, the baseline is also run on the same hardware platform with equivalent compilation flags.

The data graphs used for experiments are taken from the Stanford Network Analysis Project (SNAP) [43]. Table 6.1 summarizes the properties of these data graphs.

We use the common query graphs with central nodes that are generally used in subgraph enumeration literature. Figure 6.1 shows the query graphs used for the experiments, these queries are adapted from [1]. As a notation in Figures and Tables, we use cqx for *clique* of size x and cqxm1 for size x *clique* - 1. *Clique* is a fully connected graph.

Graph	**|V|**	**|E|**	**Max Degree**
as-skitter	1,696,415	11,095,298	35,455
soc-pokec	1,632,804	22,301,964	14,854
cit-patents	3,774,768	16,518,948	793
com-orkut	3,072,441	117,185,083	33,313
com-youtube	1,134,891	5,975,248	28,754

Table 6.1: Data Graphs used for experiments

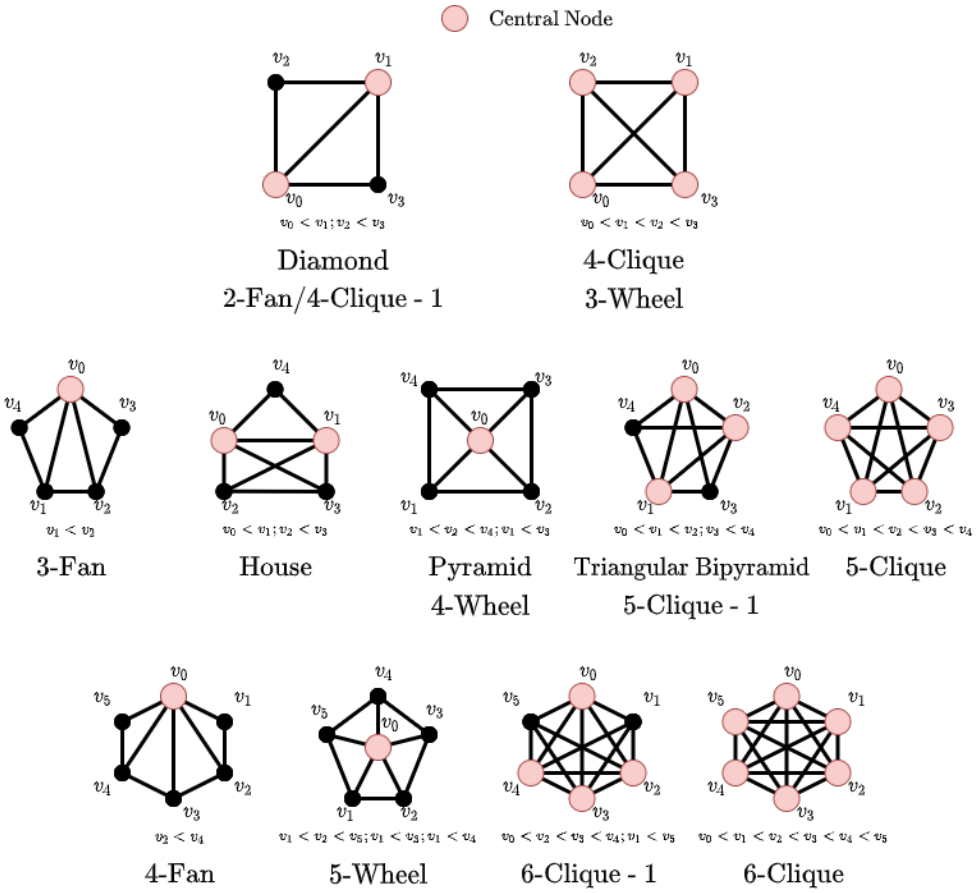


Figure 6.1: Query Graphs used for experiments

6.2 Baseline Selection

As mentioned, PARSEC [1] is used as a baseline since it is the current state-of-the-art solver in sequential, multi-core and GPU communities.

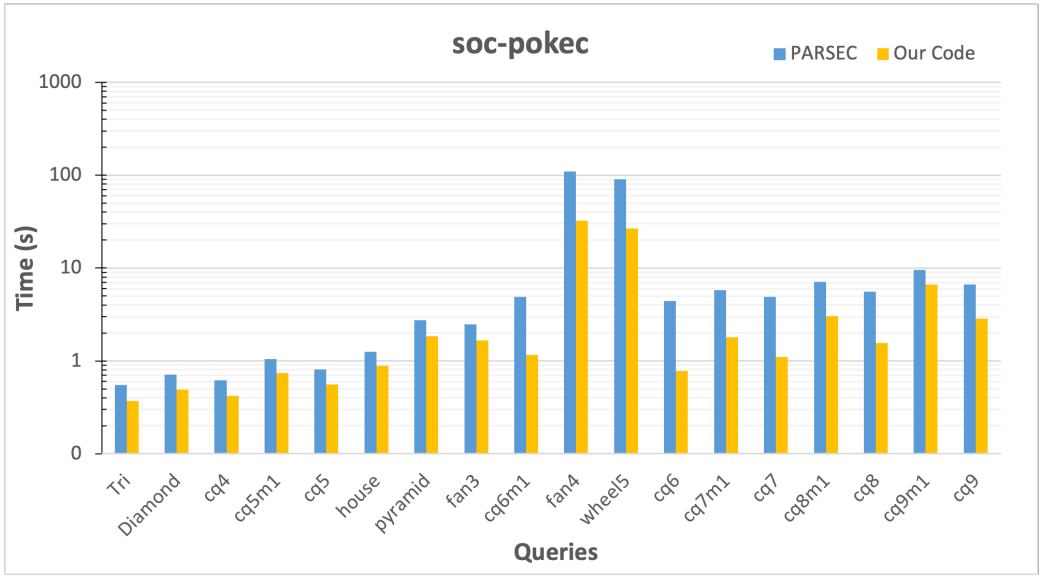
6.3 Performance Criteria

The performance is evaluated in terms of total processing time which includes query preprocessing, data graph preprocessing, and search tree traversal times. We use following rules for reporting run times:

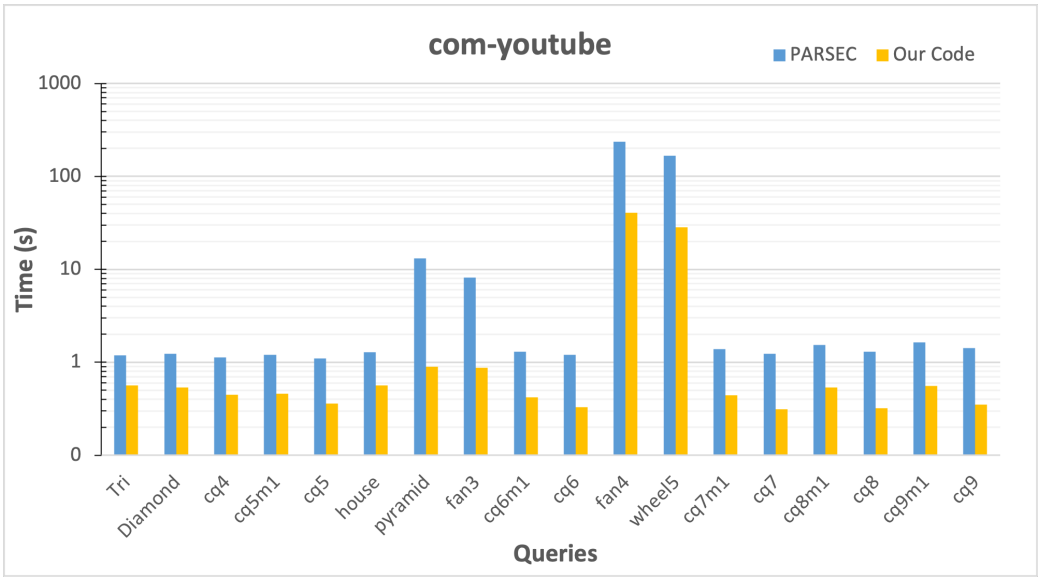
1. For instances with total run times less than 1 second, the program is executed 5 times and average times are reported.
2. Time taken for compilation, reading data graphs, and printing all enumerations is not considered for baseline and our implementation.
3. The baseline presents 2 parallelization schemes with different performances and mentions that the edge per block scheme works better than node per block for templates of size greater than five. This observation is used for fixing one parallelization scheme for comparison.
4. Instances that take more than 12,000 seconds ($\sim$ 3.3 hours) are terminated, and their times are not reported. The corresponding speedup values are reported as blank in all tables.

6.4 Final Results and Analysis

Figure 6.2 gives a comparison in runtimes across all graphs and queries on a logarithmic scale. Table 6.2 gives individual speedups and geometric mean speedups across different templates and data graphs. The highest speedup among all observations is 14.65×, while the lowest speedup is 0.73×. The geometric mean speedups across different queries are separately plotted in Figure 6.3. Roughly, the speedups increase with increasing query size for queries in similar families. Queries in the family pyramids, fans, and wheels have higher speedups as compared to others. This observation can be explained by Figure 5.6 since *fans* and *wheels* has higher load imbalance than

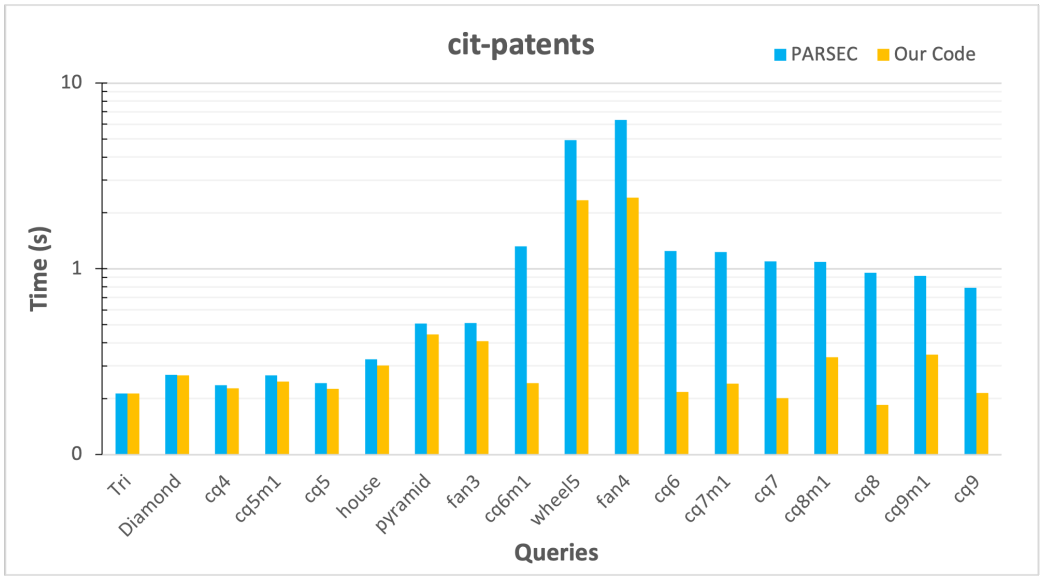


(a) Data Graph: soc-pokec

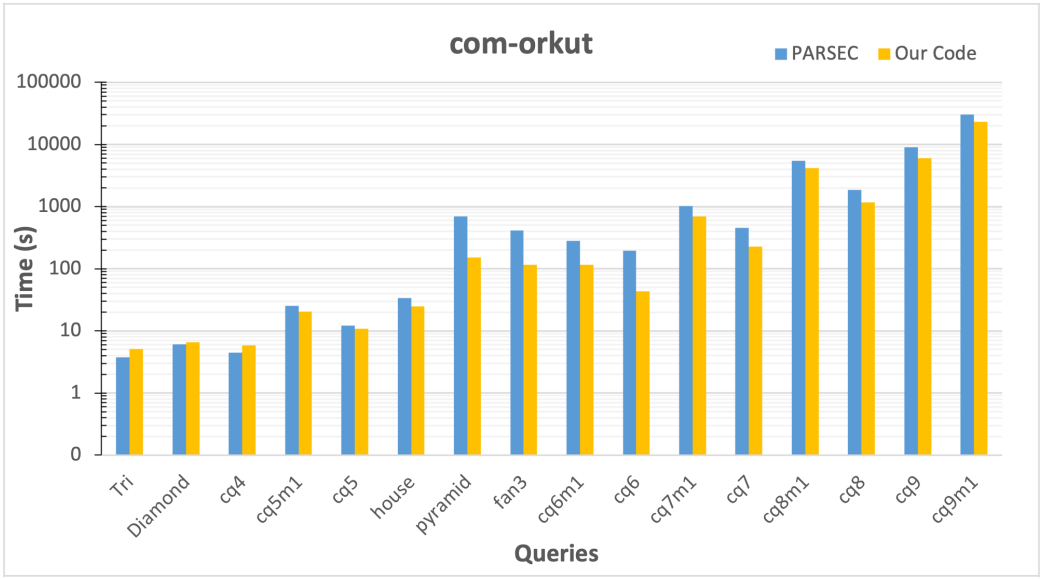


(b) Data Graph: com-youtube

Figure 6.2: Execution times and speedups compared to PARSEC [1]

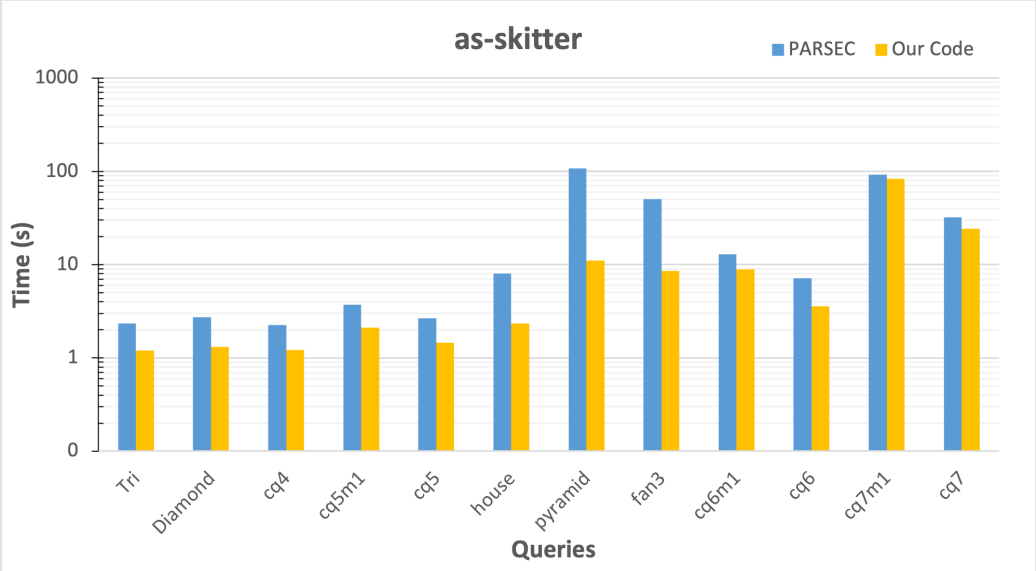


(c) Data Graph: cit-patents



(d) Data Graph: com-orkut

Figure 6.2: Execution times and speedups compared to PARSEC [1]



(e) Data Graph: as-skitter

Figure 6.2: Execution times and speedups compared to PARSEC [1]

Queries	soc-pokec	com-youtube	cit-patents	com-orkut	as-skitter	Geo Mean
Tri	1.49	2.11	1.00	0.73	1.95	1.35
Diamond	1.44	2.31	1.00	0.92	2.08	1.45
cq4	1.47	2.51	1.03	0.76	1.85	1.40
cq5m1	1.41	2.62	1.08	1.23	1.76	1.54
cq5	1.46	3.04	1.07	1.13	1.83	1.58
house	1.41	2.27	1.08	1.36	3.45	1.75
pyramid	1.50	14.65	1.15	4.5	9.75	4.06
fan3	1.50	9.38	1.26	3.59	5.89	3.27
cq6m1	4.19	3.09	5.47	2.41	1.44	3.01
fan4	3.41	5.82	2.62	–	–	3.73
wheel5	3.39	5.93	2.10	–	–	3.48
cq6	5.68	3.65	5.76	4.48	2.01	4.04
cq7m1	3.19	3.14	5.11	1.47	1.11	2.42
cq7	4.43	3.96	5.45	2.01	1.32	3.03
cq8m1	2.32	2.88	3.25	1.32	–	2.31
cq8	3.55	4.06	5.15	1.58	–	3.29
cq9m1	1.44	2.93	2.65	1.32	–	1.96
cq9	2.32	4.06	3.67	1.52	–	2.69
Geo Mean	2.25	3.74	2.20	1.61	2.29	2.33

Table 6.2: Speedups across all queries and data graphs

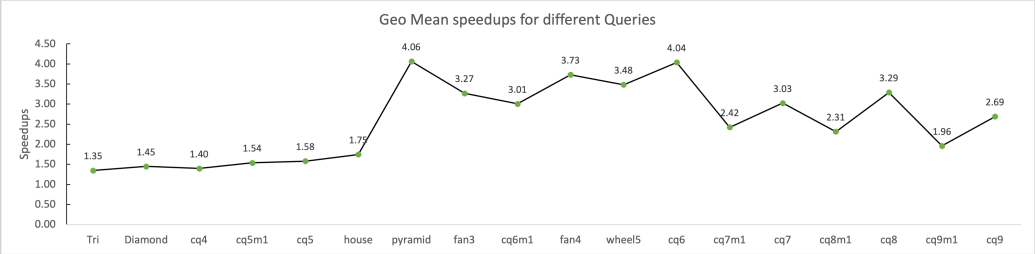


Figure 6.3: Geo Mean speedups across all data graphs

other queries. Data graphs cit-patents and com-orkut perform worse than the baseline because the inducing subgraph kernel has high load imbalance. Since the processing time for these queries is fairly small, load imbalance in the subgraph induction step is significant.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, brief review of subgraph enumeration algorithms using state space search techniques was performed. Insights from traditional sequential algorithms and understanding of GPU architecture were used to apply different improvements on the state-of-the-art subgraph enumeration application, PARSEC [1]. To reduce the number of intersections, the problem of detecting optimal reuse mappings was posed as an optimization problem and a closed form solution was found. The search space of the search tree was reduced by using a 2 phase hybrid symmetry breaking strategy which uses heuristics to dynamically decide the symmetry breaking direction. Further, an in depth analysis of PARSEC [1] was performed to point out the underlying load imbalance due to the parallelization scheme and static work split. The load balance was improved by using a hybrid parallelization scheme and splitting the kernel. These different techniques help achieve modest geometric mean speedups of up to 4× across different data graphs and up to 3.7× across all queries. These efforts also help develop numerous computational insights and unlocks scope for further improvement.

For future work, the load balance can be improved by using dynamic strategies like work sharing between blocks. The biggest drawback of this implementation is that it is only applicable for queries with a central node. Different algorithms need to be developed for adding support to more queries, one idea may be converting edges in a query graphs to vertices to support queries with central edge and try adding dummy edges to queries to create an artificial central node. [SK: Do these sound a bit vague?]

Another drawback of this implementation is the memory requirement being square of the undirected max degree. This inhibits the implementation to be truly scalable for large dense graphs. Orientation based techniques were tried which take the partially induced subgraph and find remaining candidates using binary search techniques. However, these implementations do

not perform well with increasing query size and have even worse load imbalance. But they can still be used due to smaller memory footprint and being faster than traditional sequential algorithms.

After fixing the load balance problems, the implementation can be scaled to multiple GPUs by using the CPU unified memory to store the data graph and process even bigger data graphs and larger queries in reasonable time. Graph partition packages like METIS [44] can be used to scale further by partitioning graphs and storing them to more than one CPU nodes.

[SK: Any more ideas?]

REFERENCES

[1] V. Dodeja, M. Almasri, R. Nagi, J. E. Xiong, and W. mei Hwu, "Parsec: Parallel subgraph enumeration in cuda," *IEEE*, 2022.

[2] I. Wegener, *Complexity Theory - Exploring the Limits of Efficient Algorithms*. Springer Berlin, Heidelberg, 2005.

[3] R. Alonso and S. Günnemann, "Mining contrasting quasi-clique patterns," *CoRR*, vol. abs/1810.01836, 2018. [Online]. Available: http://arxiv.org/abs/1810.01836

[4] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics (Oxford, England)*, vol. 24, no. 13, pp. i241–i249, Jul 2008. [Online]. Available: https://doi.org/10.1093/bioinformatics/btn163

[5] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Proceedings of the* 12^{th} *International Conference on Artificial Intelligence and Statistics*, D. van Dyk and M. Welling, Eds., vol. 5. PMLR, 16–18 Apr 2009, pp. 488–495.

[6] T. Milenković and N. Przulj, "Uncovering biological network function via graphlet degree signatures," *Cancer informatics*, vol. 6, pp. 257–273, 2008. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/19259413

[7] I. Gutman, C. Rücker, and G. Rücker‡, "On walks in molecular graphs," *Journal of chemical information and computer sciences*, vol. 41, pp. 739–45, 05 2001.

[8] J. Leskovec, A. Singh, and J. Kleinberg, "Patterns of influence in a recommendation network," in *Proceedings of the 10th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, ser. PAKDD'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 380–389.

[9] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[10] H. Sutter and J. Larus, "Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry." *Queue*, vol. 3, no. 7, p. 54–62, sep 2005. [Online]. Available: https://doi.org/10.1145/1095408.1095421

[11] T. Biagini, F. Petrizzelli, M. Truglio, R. Cespa, A. Barbieri, D. Capocefalo, S. Castellana, M. F. Tevy, M. Carella, and T. Mazza, "Are gaming-enabled graphic processing unit cards convenient for molecular dynamics simulation?" *Evolutionary Bioinformatics*, vol. 15, p. 1176934319850144, 2019. [Online]. Available: https://doi.org/10.1177/1176934319850144

[12] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, "A step towards energy efficient computing: Redesigning a hydrodynamic application on cpu-gpu," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, 05 2014, pp. 972–981.

[13] Business-Quant. (2022) Nvidia revenue breakdown by end-market (2017-2022). [Online]. Available: https://businessquant.com/nvidia-revenue-by-end-market

[14] P. Gupta, "Cuda refresher: The cuda programming model," NVIDIA, Tech. Rep., 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[15] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, p. 31–42, jan 1976. [Online]. Available: https://doi.org/10.1145/321921.321925

[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endow.*, vol. 1, pp. 364–375, 2008.

[17] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endow.*, vol. 8, pp. 617–628, 2015.

[18] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1199–1214. [Online]. Available: https://doi.org/10.1145/2882903.2915236

[19] C. R. Rivero and H. M. Jamil, "Efficient and scalable labeled subgraph matching using sgmatch," *Knowledge and Information Systems*, vol. 51, pp. 61–87, 2016.

[20] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble, "Sequential and parallel solution-biased search for subgraph algorithms," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds. Cham: Springer International Publishing, 2019, pp. 20–38.

[21] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artificial Intelligence*, vol. 174, no. 12, pp. 850–864, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370210000718

[22] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proc. VLDB Endow.*, vol. 8, no. 10, p. 974–985, jun 2015. [Online]. Available: https://doi.org/10.14778/2794367.2794368

[23] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proc. VLDB Endow.*, vol. 10, no. 3, p. 217–228, nov 2016. [Online]. Available: https://doi.org/10.14778/3021924.3021937

[24] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 6 2020, pp. 1083–1098.

[25] P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *Proc of the 3rd IAPR-TC-15 International Workshop on Graph-based Representation*, 01 2001.

[26] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[27] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 804–818, 2018.

[28] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *Annual International Conference on Research in Computational Molecular Biology*. Springer, 2007, pp. 92–106.

[29] A. for Computing Machinery. Special Interest Group on Management of Data., *Turbo-ISO*. ACM, 2013.

[30] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endow.*, vol. 8, no. 5, p. 617–628, jan 2015. [Online]. Available: https://doi.org/10.14778/2735479.2735493

[31] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 625–636. [Online]. Available: https://doi.org/10.1145/2588555.2588557

[32] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1447–1462. [Online]. Available: https://doi.org/10.1145/3299869.3300086

[33] S. Sun, Y. Che, L. Wang, and Q. Luo, "Efficient parallel subgraph enumeration on a single machine," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 232–243.

[34] D. Mawhirter, S. Reinehr, W. Han, N. Fields, M. Claver, C. Holmes, J. McClurg, T. Liu, and B. Wu, "Dryadic: Flexible and fast graph pattern matching at scale," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 289–303.

[35] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Database Systems for Advanced Applications*, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 299–315.

[36] W. Guo, Y. Li, and L. Tan, "Exploiting reuse for gpu subgraph enumeration," *IEEE Transactions on Knowledge and Data Engineering*, vol. PP, pp. 1–1, 11 2020.

[37] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 6 2020, pp. 1067–1082.

[38] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[39] D. Merril. (2016) Cub. [Online]. Available: https://nvlabs.github.io/cub/index.html

[40] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W. mei Hwu, "Accelerating k-clique counting on GPUs," *arXiv preprint arXiv:2104.13209*, 2021.

[41] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," in *KR*, 03 1997.

[42] S. Zampelli, Y. Deville, and P. Dupont, *Symmetry Breaking in Subgraph Pattern Matching*. John Wiley & Sons, Ltd, 2007, ch. 10, pp. 203–218. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470612309.ch10

[43] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," 2014.

[44] G. Karypis and V. Kumar, *METIS—A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices*, 01 1997. [Online]. Available: https://hdl.handle.net/11299/215346