OPTIMIZING GRAPH ANALYSES ON GPUS

by

Wei Han
Copyright by Wei Han 2021

All Rights Reserved
A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science).

Golden, Colorado

Date ____________________________

Signed: __________________________

Wei Han

Signed: __________________________

Dr. Bo Wu
Thesis Advisor

Golden, Colorado

Date ____________________________

Signed: __________________________

Dr. Tracy Camp
Professor and Head
Department of Computer Science
ABSTRACT

The massive parallelism on GPUs provides opportunities to accelerate computation on large real-world graphs. However, implementing parallel graph algorithms on GPUs is quite challenging because of the following two facts. First, the DRAM on a GPU is limited and the size of some real-world graphs are possibly larger than the global memory on a GPU. Second, the irregularity of node degrees poses severe load balancing issues on graph applications. We address these issues in two main graph problems in this thesis.

In the first part of this thesis, we introduce a new GPU-based graph traversal system, called Graphie, which can handle out-of-core graphs. We divide the edge list of a graph into partitions with equal number of edges so that each partition can fit into the global memory. The biggest advantage of this approach is that it improves load balancing. Then we propose several techniques to leverage features on modern GPUs to accelerate graph traversal algorithms. We develop two renaming techniques to quickly find updated partitions and cache vertex values in the shared memory for lower memory latency. We use a small array to keep track of active partitions and to guide the data movement between the host and device where unnecessary data transfers are avoided. We further utilize the Hyper-Q technique on modern GPUs to overlap the computation and data movement.

In the second part, we present our GPU-based subgraph matching system, DGSM. Current subgraph matching frameworks suffer from the need to maintain overly large intermediate states, which is normally far larger than a GPU’s main memory. As a result, these frameworks cannot match large queries on large data graphs. Another issue is that there is redundant computation in their systems. DGSM can solve these two issues in the existing subgraph matching systems. We design a GPU friendly data structure to reduce memory access latency. Several techniques are incorporated in our system to further improve the performance. Our experimental results show that our system is about 2 orders of magnitude faster than the state-of-the-arts systems on both labeled and unlabeled subgraph matching.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................ iii

LIST OF FIGURES .............................................................................................. vii

LIST OF TABLES ................................................................................................ x

ACKNOWLEDGMENTS ......................................................................................... xi

CHAPTER 1 INTRODUCTION ................................................................................ 1

1.1 Motivations ...................................................................................................... 1

1.2 Challenges in GPU-based Graph Frameworks .............................................. 2

1.2.1 Challenge I: Limited main memory ........................................................... 2

1.2.2 Challenge II: Load Imbalance .................................................................... 3

1.2.3 Challenge III: Random Accesses ............................................................... 5

1.2.4 Challenge IV: Hardware Thread Scheduling ............................................. 5

1.2.5 Challenge V: Redundant Storage and Computation .................................. 6

1.3 Dissertation Organization ............................................................................. 7

CHAPTER 2 BACKGROUND ............................................................................... 8

2.1 Definition ......................................................................................................... 8

2.2 Graph Representations ................................................................................... 10

2.3 Important Graph Algorithms ......................................................................... 13

2.4 Large Scale Graph Analytic Models .............................................................. 14

2.5 GPU Architecture .......................................................................................... 16

CHAPTER 3 GRAPHIE: GRAPH TRAVERSALS ON A GPU ......................... 24

3.1 Introduction ..................................................................................................... 25

3.2 Background and Motivation ......................................................................... 28

3.2.1 Graph processing models and data organization ....................................... 28

3.2.2 Out-of-memory graph traversals on GPUs .............................................. 29

3.3 Optimizing Kernel Execution through Vertex Renaming ............................ 34

3.3.1 Improving GPU memory access efficiency ............................................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Efficiently Activating Partitions</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Minimizing Data Transfer Overhead through Asynchronous Streaming</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Graphie System</td>
<td>44</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Graphie workflow</td>
<td>44</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Programming interface</td>
<td>46</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Selecting partition size</td>
<td>46</td>
</tr>
<tr>
<td>3.6</td>
<td>Experimental Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Experiment setting</td>
<td>47</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Overall results</td>
<td>49</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Breakdown of the optimization benefits</td>
<td>52</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Results on asynchronous streaming</td>
<td>53</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Overhead of the renaming processes</td>
<td>54</td>
</tr>
<tr>
<td>3.7</td>
<td>Related Work</td>
<td>54</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Background</td>
<td>70</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Subgraph Matching</td>
<td>70</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Two types of embedding</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>Related Work</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Motivations</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Challenges in GPU-based subgraph isomorphism</td>
<td>80</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Challenge I</td>
<td>80</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Challenge II</td>
<td>80</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Challenge III</td>
<td>80</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Challenge IV</td>
<td>81</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Challenge V</td>
<td>81</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>Power Law Distribution of Degrees</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Symmetries in Clique 4</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>A Illustration of A Social Network with 6 Persons</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Two types of Graph</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Caption</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Adjacency Matrix</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Adjacency List</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>CSR</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>Edge List</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>An illustration of the GAS Model</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>The Filter-join Model in Subgraph Matching</td>
<td>16</td>
</tr>
<tr>
<td>2.10</td>
<td>Illustration of GPU Architecture</td>
<td>17</td>
</tr>
<tr>
<td>2.11</td>
<td>Memory Hierarchy</td>
<td>18</td>
</tr>
<tr>
<td>2.12</td>
<td>Thread Hierarchy</td>
<td>19</td>
</tr>
<tr>
<td>2.13</td>
<td>Aligned and Misaligned Memory Transactions</td>
<td>20</td>
</tr>
<tr>
<td>2.14</td>
<td>Uncoalesced Memory Transaction</td>
<td>20</td>
</tr>
<tr>
<td>2.15</td>
<td>Timeline of CUDA Streams</td>
<td>22</td>
</tr>
<tr>
<td>2.16</td>
<td>False Dependency</td>
<td>23</td>
</tr>
<tr>
<td>2.17</td>
<td>False Dependency</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>An example graph and its partitioned edge list</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustration of the renaming process</td>
<td>36</td>
</tr>
</tbody>
</table>
Figure 4.17 Speedup of 5 queries in Motif-4 ............................................ 101
Figure 4.18 Speedup with DAF for Labeled Queries ................................. 103
Figure 4.19 Speedup of Labeled Queries on Amazon with GSI .................. 104
Figure 4.20 Speedup of Two Queries with GSI .......................................... 104
LIST OF TABLES

Table 3.1  Datasets Used in The Experiments .................................................. 30
Table 3.2  GPU Specifications ........................................................................... 47
Table 3.3  Execution times of Graphie and the compared systems....................... 50
Table 3.4  The overhead of renaming in seconds. ................................................ 54
Table 4.1  Statistics of Datasets ........................................................................ 93
Table 4.2  Baseline Performance(Second) .......................................................... 94
Table 4.3  Edge-induced Matching Performance(ms) .......................................... 96
Table 4.4  Vertex-induced Matching Performance(ms) ........................................ 98
Table 4.5  Performance with threshold (ms) ..................................................... 99
Table 4.6  Clique Performance(s) ................................................................. 101
Table 4.7  Clique Performance(s) ................................................................. 101
Table 4.8  Performance of 6 Labeled Patterns(ms) ............................................ 102
Table 4.9  Performance of 5 Labeled Patterns(ms) ............................................ 102
Table 4.10 Labeled Performance with Threshold(ms) ....................................... 102
ACKNOWLEDGMENTS

This thesis would not be possible without help of my advisor, colleagues, friends, and my family. First and foremost, I would like to thank my advisor Dr. Bo Wu for encouraging me when I am feeling low, introducing me to the high performance computing society, and teaching me how to be a good researcher. It is my greatest honor to work with him toward my PhD. I would like to express my gratitude to dissertation committee members Dr. Tracy Camp, Dr. Chuan Yue, Dr. Xiaoli Zhang, and Dr. Bin Ren for sitting on my committee and their valuable feedback and support. I am lucky to have them guide my research. I would like to thank our group members, Daniel Mawhirter, Connor Holmes, and Akshit Sharma, who are always willing to help me during my PhD study. I would also like to thank my family for their endless help and support throughout my life. Last and most importantly, I would like to thank the CS department and National Science Foundation for their financial support in the course of pursuing my PhD.
CHAPTER 1
INTRODUCTION

In this chapter, we demonstrate why it is necessary and urgent for computer scientists to develop a fast and scalable graph processing system and present reasons why GPUs are suitable for graph algorithms. We also discuss challenges of implementing an efficient GPU-based graph processing system. We end this chapter with the outline of this thesis.

1.1 Motivations

Graph is a fundamental and ubiquitous data structure that has been applied to many scientific domains including logistical networks, social networks, artificial intelligence, chemical engineering, and bioinformatics [13, 16, 20, 45]. Graphs are applied across many fields, from biology to neural network. Biologists use graphs to represent various types of biological interactions. Through the analysis on the biological graphs, they can understand the structure of molecules and their interactions. Netflix would like to find all the cliques with $k$ vertices in its social network to promote an advertisement. Over the past few years, some graph algorithms were applied to deep learning methods. Solving these problems depends heavily on graph algorithms. Graph algorithms are usually the bottleneck of the performance of these problems.

The performance of graph algorithms is exacerbated in the big-data era. The size of real-world graphs is getting bigger and bigger since the beginning of this millennium. It is common nowadays that there are millions of vertices and billions of edges in a graph. However, most users want to get real-time responses of graph algorithms on a large scale graph. Therefore, efficient algorithms on large scale real-world graphs are demanding and attract many computer scientists’ attention. The two popular computational paradigms in graph processing are vertex-centric [32] and edge-centric [59] programming models. They lead to the fact that a real-world graph presents plenty of parallelism in the two programming models. The larger a graph is, the more data parallelism exists in graph applications. How to exploit massive data parallelism in a graph becomes critical to
improve the performance of graph applications. As far as enormous data parallelism is concerned, GPUs come in handy to accelerate these applications.

GPU stands for Graphics Processing Unit and is a throughput-oriented processor designed for parallel computing. A GPU is normally equipped with many streaming processors (SM) where thousands of threads could be launched simultaneously. NVIDIA revolutionized the world of high performance computing by introducing the Compute Unified Device Architecture (CUDA) programming language in 2007, which is used to program on GPUs. The advent of CUDA eases the programming efforts and makes the implementation of a parallel application as easy as writing a C program. Therefore, a GPU device becomes the mainstream to serve as an application accelerator when massive parallelism presents in the application. A plethora of parallelisms in a graph make GPU a suitable platform for graph applications. Although we can take advantage of parallelism in a GPU to accelerate graph applications, writing efficient parallel graph algorithms still remains open since the underlying architecture of GPUs imposes restrictions and challenges for graph processing.

### 1.2 Challenges in GPU-based Graph Frameworks

A GPU-based graph system is more complicated than CPU-based counterparts because of the special architecture features on GPU devices. These features are double edged swords and provide both challenges and opportunities to GPU applications. In this section, we will discuss several challenges when implementing GPU-based graph frameworks.

#### 1.2.1 Challenge I: Limited main memory

A GPU has a separate high bandwidth HBM2 or GDDR5 DRAM from its host. In general, the main memory on GPU is far less than the main memory on the host. The typical size of a GPU’s main memory is usually less than 20GB. It puts a limit on the size of the graph that can be kept on a GPU device and processed by the GPU. In reality, some real-world social graphs like Friendster and Twitter are too big to be stored on a GPU. Even if a graph can fit in the GPU’s main memory, some algorithms, like graph mining
and subgraph isomorphism, may generate a huge amount of intermediate data which can possibly exceed the size of the main memory. Although we can use the host memory as a backup storage, it will lead to a tremendous performance drop due to the relatively low communication bandwidth between the host and the device. This limitation poses the greatest challenge for subgraph isomorphism as the intermediate data can reach up to many TBs, which far exceeds the size of a CPU’s main memory let alone the GPU global memory.

1.2.2 Challenge II: Load Imbalance

The inherent divergence of vertex degrees in a graph poses a notorious load imbalance problem to GPU-based graph applications. In most real-world graphs, the size of a vertex neighbor list follows the power law where only a small portion of vertices has a very large number of neighbors. Figure 1.1 shows the power law distribution of degrees of two social networks, Orkut and Livejournal. It is obvious in Figure 1.1 that the majority of vertices has degrees less than 10. The power law distribution results in severe load imbalancing and performance degradation in the vertex-centric computation model where each thread is mapped to the computation on a vertex. The more neighbors a vertex has, the more work is needed for that vertex. These hot vertices become the bottleneck of graph analytics in vertex-centric model. The same issue also presents in subgraph isomorphism, and it might be worse than that in graph traversal algorithms. The load imbalance further leads to the hardware underutilization. Therefore, the load imbalancing is an inevitable problem for designing efficient graph algorithms.
Figure 1.1: Power Law Distribution of Degrees

(a) Orkut

(b) Livejournal

Figure 1.1: Power Law Distribution of Degrees
1.2.3 Challenge III: Random Accesses

For most GPU applications, most data accesses start from the global memory. It is, therefore, critical to maximize the global memory bandwidth for a kernel’s performance. Unlike the CPU cache which is designed for both spatial and temporal locality, the GPU cache system is optimized for spatial locality but not temporal locality. To achieve the best performance, global memory accesses must be aligned and coalesced. However, random memory accesses are unavoidable and dominate memory transactions in most graph applications. For example, the computation in graph traversals will depend on the value of a vertex’s neighbors. These vertex values are generally scattered in the memory. So the hot vertices have the worst random accesses. The random accesses also exist in subgraph isomorphism, and we will give a detailed discussion about this issue in the second half of this thesis. The random accesses will incur more memory transactions and waste memory bandwidth. If the random accesses in graph algorithms are not properly taken care of, they will render other optimization useless since most graph algorithms are memory-bound. Although it is impossible to completely remove random accesses in graph algorithms, we will limit the random accesses to a large extent without introducing memory overhead.

1.2.4 Challenge IV: Hardware Thread Scheduling

Although massive data parallelism is supported on a GPU device, the actual parallelism is determined by hardware resource partitions. This actual parallelism is called thread occupancy. The thread occupancy is governed by multiple factors, for example thread register usage and shared memory usage by a thread block. These hardware resources are scarce and partitioned among thread blocks on an SM. If a kernel oversubscribes register, it will either limit the thread occupancy on an SM or incur higher read/write memory latency due to register spilling. In addition, the concurrent thread blocks on an SM share the same memory pool. If one thread block consumes too much shared memory, the number of thread blocks scheduled on an SM will be reduced. Both cases can potentially reduce thread occupancy significantly. These two factors need to be
taken into account in order to improve the data parallelism of an algorithm.

1.2.5 Challenge V: Redundant Storage and Computation

Redundant storage and computation are typical issues in subgraph isomorphism. The cause of redundant storage comes from the symmetries in a pattern graph, while the redundant computation originates from the symmetries, precomputing memory consumption, and connectivity check. Here a symmetry means that a permutation of vertices leaves the pattern invariant.

![Symmetries in Clique 4](image)

Figure 1.2: Symmetries in Clique 4

Please see the Figure 1.2 for one symmetry out of 24 in clique-4 where any permutation on (0, 1, 2, 3) will leave the pattern invariant. Here numbers denote vertex i.d. and capital letter A stands for the label of a vertex. It will lead to 23 redundant copies for the same pattern. The redundancies waste not only memory space but also hardware resources. One possible solution to symmetrical redundancies is introducing an automorphism test. However, there is no efficient parallel automorphism test available for GPUs, and it is quite expensive to do automorphism tests on GPUs. Implementing an automorphism test in a GPU-based subgraph isomorphism system may outweigh keeping these redundancies throughout the computation and is not the best solution. In addition, the duplicate computation for memory consumption can be avoided by conservatively allocating memory in the main memory. For example, there are 100 instances in the current step. We can
preallocate $D$ slots for each instance, where $D$ is the maximum degree in the data graph because the current frameworks adopt the bulk synchronous parallel model and breath-first exploration strategy. However, this approach is not realistic in most frameworks since tons of instances need to be materialized in the main memory for a real-world graph. Such approach also wastes a lot of memory space since only the majority of vertices has a few neighbors. These are the common issues in other frameworks. So these redundancies limit their performance and restrict them to small patterns or graphs with plenty of labels. However, they provide us opportunities to optimize GPU-based subgraph isomorphism, and we will revisit them in the Chapter 4.

The main purpose of this thesis is to address these notorious challenges in GPU-based graph traversal and subgraph isomorphism algorithms.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 gives an introduction to the concepts in graph theory which will be needed in this thesis and then present an overview of GPU architecture and important features which are critical to the performance of a system. Chapter 3 will discuss difficulties in implementing efficient out-of-core graph traversal algorithms and present our solution, an edge-centric graph traversal framework named Graphie, to these issues. Chapter 4 will explore another fundamental problem in graph analytics, subgraph isomorphism. Chapter 4 also presents our solutions to several challenges including the redundant computation, large intermediate data, coalesced memory accesses, and load balance in subgraph isomorphism. Finally, Chapter 5 concludes the thesis by providing a summary of our works and the directions for future work.
Before we dive into the main topics of this thesis, we will first present the concepts and notations of the graph data structure used in the thesis. Then we will give a brief introduction to the GPU architecture.

2.1 Definition

The graph is a flexible data structure and can be used to model any problem which involves discrete objects and relationships among these objects. An obvious instance is a Facebook social network where each person is abstracted as a node and friendship between two persons is denoted by an edge connecting the two nodes. Figure 2.1 shows a social network of 6 persons. For convenience, we have assigned a number or id (0, · · · , 5) to distinguish each node in the figure.

Generally speaking, we can define a graph as a 4-tuple \((V, E, \Sigma, l)\), where \(V\) is the vertex set \(V = \{v_1, \ldots, v_m\}\), \(E\) stands for the edge set \(E = \{e_1, \ldots, e_n\} \subseteq V \times V\), \(\Sigma\) is a finite discrete set, and \(l\) is the labeling function which maps each vertex to a unique label in the set \(\Sigma\). If there is only one element in the set \(\Sigma\), the last two components can be removed. We call such a graph as an unlabeled graph. Without loss of generality, we only consider the case where only vertices have been labeled. All the algorithms can be easily extended to graphs whose edges have labels. In the above example, the vertex set is
The edge set is \{(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (2, 3), (3, 4), (4, 5)\}. The label set is \{Don, Ali, Bob, Joe, Ian, Ted\}. The labeling function \(l\) is defined by

\[
\begin{align*}
0 & \mapsto \text{Don} \\
1 & \mapsto \text{Ali} \\
2 & \mapsto \text{Bob} \\
3 & \mapsto \text{Joe} \\
4 & \mapsto \text{Ian} \\
5 & \mapsto \text{Ted}. 
\end{align*}
\]  

Each element \(e_i\) in the edge set is a pair of vertices \((u_i, v_i)\). For later convenience, we use \(N(E)\) and \(N(V)\) to denote the number of edges and vertices in a graph. The neighbors of a vertex \(v \in V(G)\) is represented by \(N_G(v)\). The degree of a vertex \(d_G(v)\) is the size of that vertex’s neighbor list \(N_G(v)\).

A graph is directed if there is an incidence function \(\Psi\) which associates each edge with an order pair of vertices in \(V\). The incidence function indicates there is an orientation on an edge. The order \(u_i\) and \(v_i\) are not interchangeable. The two end nodes of an edge, \(u_i\) and \(v_i\), are called the source and destination vertices of the edge, respectively. We also call the edge an out-going edge of vertex \(u_i\) and an incoming edge of \(v_i\). In a directed graph, the degree of a vertex is split into two categories, the out-going degree and incoming degree. A directed graph is also called digraph. A directed graph with no cycles is named directed acyclic graph or DAG. Figure 2.2b is not a DAG because there is a cycle path \(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1\) in the figure.

If the edges represent nothing more than connectivity or the order of a pair does not matter, a graph is named an undirected graph. In another word, the edge is bidirectional. The difference between directed and undirected graphs can be illustrated by the following example. If the vertices represent cities in the U.S. and the edges denote the two-way roads connecting two cities, the graph is undirected because city \(A\) has a pathway to city \(B\) only if city \(B\) has a pathway to city \(A\) as well. If some roads correspond to one-way roads, the graph is directed because they don’t have reciprocals. Figure 2.2 shows pictorial representation of both directed and undirected graphs.

If there are multiple edges connecting the same two vertices in a graph (both directed and undirected), the graph is called multi-graph. In this thesis, we assume that graphs of
interest are not multi-graphs and there are no self-loops in a graph by default.

![Two types of Graph](image)

**Figure 2.2: Two types of Graph**

A weighted graph is defined as a graph where there is a weight function which maps each edge in the edge set $E$ to a real number, that is, $w : E \rightarrow \mathbb{R}$. In the context of this thesis, we assume that edge weights are non-negative real numbers. A weighted graph can be either a directed graph or an undirected graph. The weight of an edge is often referred to as the cost of the edge. In graph traversal algorithms, the weight may be a measure of the length of a route between neighbor vertices or the capacity of the route.

A graph is called connected if every node can reach every other node in the graph. A graph which is not connected can be decomposed into several connected components, which are maximal connected subgraphs. Figure 2.3a displays the examples of connected and unconnected graphs.

### 2.2 Graph Representations

The performance of a graph algorithm generally depends on how it is stored in the computer. A graph is commonly presented in the following five ways, adjacency matrix, adjacency list, compressed sparse row (CSR), compress sparse column (CSC), and edge list. Each representation has both advantages and disadvantages. The choice depends on the algorithm of concern.
For example, page rank defined by Equation 2.2 prefers CSC over the adjacency matrix because the value of a node depends on all the edges incident to the node. But the breath-first search algorithm favors CSR rather than the other representations. In other words, pull-based algorithms prefer CSC, while push-based algorithms prefer CSR.

\[ PR(u) = \sum_{v \in N(u)} \frac{PR(v)}{D(v)}, \]  

(2.2)

The natural and simple way to store a graph is the adjacency matrix format. If a node \( u \) with id \( i \) is connected to a node \( v \) with id \( j \), the entry on the \( i \)th row and \( j \)th column of the adjacency matrix is set to one. Otherwise, the value is 0. Figure 2.4 displays the adjacency matrix representation of the graph 2.1.

The adjacency matrix representation takes \( N(V)^2 \) space overhead to store the edge information. It takes \( O(1) \) time to determine whether a node \( u \) has an edge connecting another node \( v \). If a graph is undirected, its adjacency matrix is symmetrical along the
diagonal. So we only need to store the upper triangle of the matrix. It will reduce the space overhead down to $N(V)^2/2$.

\[
\begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
\]

Figure 2.4: Adjacency Matrix

In reality, most graphs have more edges than vertices and their adjacency matrices are sparse. It means that we use plenty of space to store 0s in the adjacency matrix. Therefore, the disadvantage of the adjacency matrix is that we waste a lot of space to store 0s if a graph is sparse.

The adjacency list representation is a natural alternative to optimize the space overhead of the adjacency matrix. The adjacency list maintains a vertex-indexed array of lists of the vertices adjacent to each other. All the zero elements in the adjacency matrix are removed in the adjacent list. The Adjacency list reduces the space overhead down to $O(E + V)$ at the cost of more expensive connectivity check than the adjacency matrix. The connectivity check takes $O(1)$ time in the adjacency matrix, while it takes to $O(D)$ or $O(\log(D))$ in the adjacency list. Here $D$ is the maximum degree of a graph. The adjacency list representation of Figure 2.1 is shown in Figure 2.5.

![Figure 2.5: Adjacency List](image)

The CSR and CSC are the most compact ways to store the information of a graph. The
CSR and CSC representations consist of two arrays. One array is used to store the neighbors of a vertex. And all the neighbors of a vertex are located in a continuous chunk in the array. This array has all the connectivity information of the graph. The other array is the index offset array of a node. The array tells where the neighbors of a vertex are located in the edge array.

\[
\text{offset: } [0, 4, 7, 10, 13, 17, 18, 19] \\
\text{dst: } [1, 2, 3, 4, 0, 2, 4, 0, 1, 3, 0, 2, 4, 0, 1, 3, 5, 4]
\]

Figure 2.6: CSR

It takes \(O(D)\) or \(O(\log D)\) to determine if two vertices are connected or not. The space overhead of the CSR or CSC is \(O(E + V)\). The CSR is normally used in the push-based computation. On the contrary, the CSC is the ideal representation for a pull-based algorithm.

The edge list representation has two arrays. The source and destination identifications of the edges are stored in separate arrays. Therefore, the space overhead of the edge list is \(O(E)\). The benefit of the edge list representation is that it takes \(O(1)\) time to figure out the source id of an edge with a given offset index in the edge array. The downside of the edge list is that more space is used to save a vertex id multiple times, if that vertex has more than one neighbors. Figure 2.7 presents the edge list of the graph Figure 2.1.

\[
\text{src: } [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5] \\
\text{dst: } [1, 2, 3, 4, 0, 2, 4, 0, 1, 3, 0, 2, 4, 0, 1, 3, 5, 4]
\]

Figure 2.7: Edge List

### 2.3 Important Graph Algorithms

Although there are plenty of interesting graph problems, we will touch on a few fundamental graph algorithms. Many other algorithms can be solved by varying these more significant ones. In this thesis, we will study the following backbone algorithms.
Breath First Search (BFS): Given a graph $G = (V, E)$ and a source vertex $s$, BFS can answer what vertices can be reached from the source $s$ and what the shortest hops are to these vertices.

Single Source Shortest Path (SSSP): Given a weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weight and a source vertex, SSSP will find the shortest path for the source vertex to each vertex which can be reached from $s$.

Connected Components (CC): Given an undirected graph $G = (V, E)$, CC is to find all the equivalent classes where every vertex in a class can reach every other vertex in the same class.

Subgraph Isomorphism: Given a labeled graph $G = (V, E, \Sigma, l)$ and a pattern graph $P = (V', E', \Sigma, l')$, the goal of subgraph isomorphism is to list all the subgraphs in $G$ which have the same topology as pattern graph $P$.

2.4 Large Scale Graph Analytic Models

Most parallel large scale graph computation algorithms adopt the iterative Bulk Synchronous Parallel (BSP) programming model [69]. The BSP programming model consists of three components:

1. **Communication**: A processor needs to exchange data with other processors to get the most updated values of its local partition for the computation. This step can be overlapped with the computation.

2. **Concurrent Computation**: Each processor will perform concurrent computation on its local partition.

3. **Synchronous Barrier**: Each processor needs to wait for the other processors finishing their local computation in order to proceed to the next iteration.

The major advantage of the BSP is that it offers a large amount parallelism in a graph algorithm to saturate a GPU device. However, the coarse-grained synchronizations can have a significant impact on the performance if load imbalancing presents in an algorithm.

Based upon BSP, Powergraph introduced a Gather-Apply-Scatter (GAS) model [18, 61] for graph computation. As the name indicates, the GAS model is also made up of three
phases: Gather, Apply, and Scatter.

1. **Gather:** Each vertex collects values from its neighbors since the value of a vertex depends on its neighbors in graph computation.

2. **Apply:** The value of the vertex is updated by performing a reduction on values gathered from its neighbor.

3. **Scatter:** The new state of the vertex is propagated to its neighbors through the edges emanated from the vertex.

Figure 2.8 illustrates the three stages in the GAS model. For simplicity and clarity, a directed graph is used in the illustration.

One advantage of GAS is that the asynchronous update function in the Apply phase in the GAS model generally converges faster than the BSP model.

In contrast to the three phases in graph computation models, the scale subgraph isomorphism favors a two-phase filtering and joining (or verification) model. In the filtering phase, a set of candidate vertices in the input data graph is collected for a query vertex to be matched. The joining phase extends the subpatterns in the current iteration by joining the candidate vertices from the filtering phase. Most matching frameworks will need expensive isomorphism and automorphism tests to obtain the desired pattern and to remove the duplicates. Figure 2.9 clarifies the matching process where CS and SP stand for candidate set and subpattern, respectively.
Large graph analytics can also be classified into four categories along two dimensions, CPU vs. GPU and Single vs. Distributed. In this thesis, we focus on single GPU-based graph analytic frameworks for graph computation and subgraph matching.

2.5 GPU Architecture

GPUs have become popular as application accelerators in the high performance community since NVIDIA introduced a parallel computing platform and application programming interface (API) model named CUDA in 2007. Without knowing the details of GPU architecture, it would not be possible for someone to design an efficient application. Therefore, this section will give a brief introduction to the special features of a modern NVIDIA GPU.

Figure 2.10 sketches NVIDIA GPU architecture. A GPU device is usually packed with an array of streaming processors (SMs). These SMs enable massive threads to be launched simultaneously on a device, and share the same $L2$ cache. However, each SM has its own $L1/Shared$ memory pool and register files shared by thread blocks running on it. The code running on a device is named kernel.

When a kernel is invoked from the host, the user specifies the execution configuration which tells the way of thread organization. This configuration is a two-level hierarchy and specifies the number of threads in one thread block and the number of thread blocks in a grid.
Figure 2.12 demonstrates the thread hierarchy of a kernel. Although both grid and block configuration are 3D vectors \((id_x, id_y, id_z)\) in general, 2D organizations are shown in the figure for simplicity. All the threads in the kernel are collectively called a grid. Threads in a thread block are further partitioned into a group of 32 threads called a warp. A warp is the basic execution unit on an SM. The threads in a warp execute in the fashion of SIMT. It means that all the threads in a warp must execute identical instructions on the same cycle. This could lead to warp divergence if threads in the same warp take different execution paths.

<table>
<thead>
<tr>
<th>Code Snippet 1: Warp Divergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if laneid &lt; 16 then</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3 else</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Warp divergence can severely degrade the performance of an application, because the
warp will serially execute different execution paths by disabling threads which do not take the path. The Algorithm 1 displays warp divergence. The warp divergence will reduce the amount of parallelism and should be avoided as much as possible for better performance.

Like CPUs, GPUs have multiple levels of memory hierarchy to exploit data locality. Figure 2.11 shows the memory hierarchy of a GPU. In the figure, we consider host memory as part of GPU memory hierarchy because Unified Memory has been introduced to simplify memory management since CUDA 6.0. The size and latency of each hierarchy decreases from the top to the bottom.
The global memory is the largest, highest-latency memory component and is accessible by all the threads in the grid. The global memory will be shared by all the SMs on a GPU.
Most data access starts from the global memory. As mentioned earlier, thousands of threads are supported on a device and they are hungry for data. The memory bandwidth should be large enough to keep thousands of device threads busy. In reality, the performance of most GPU applications is limited by the memory bandwidth. Therefore, maximizing global memory bandwidth is critical to a kernel’s performance. The global memory bandwidth is determined by two factors, aligned and coalesced memory accesses. Aligned memory accesses means that the first address of memory transaction is the a multiple of the cache granularity. A misaligned memory transaction will waste memory bandwidth. Figure 2.13 clarifies the aligned and misaligned memory accesses. Half of the memory bandwidth is wasted in Figure 2.13b.

A coalesced memory access refers to the 32 threads in a warp accessing a continuous memory chunk. Figure 2.13a is also an example of coalesced memory transaction, while Figure 2.14 demonstrates an uncoalesced memory access where $\frac{2}{3}$ memory bandwidth is wasted. In addition to the global memory, the $L2$ cache is shared among SMs as well.
A GPU also exposes an on-chip memory with the latency as low as $L_1$ cache to a programmer, named shared memory. The shared memory can be considered a programmable cache. On a V100 GPU, each SM has a 128KB on-chip memory which can be partitioned between $L_1$ and the shared memory. The configurable size of shared memory ranges from 0 to 96KB per SM. The shared memory is allocated per thread block and all the threads in a thread block can access the same shared memory. Hence the shared memory can be used as a program-managed cache to reduce global memory access latency if such variables will be accessed multiple times throughout the kernel and provide an intra-block thread communication mechanism among threads in a thread block. The shared memory latency is about 100X faster than the global memory latency. Therefore, the shared memory is a key component on GPU for writing well-optimized GPU-based applications.

The fastest memory space on a GPU is register. The thread blocks on an SM shared one register file. The variables declared in a kernel will automatically reside in registers. Register variables are thread private. The registers are scarce resources on a GPU. If a kernel uses too many thread private registers, the excess registers will spill to local memory, also called register spilling. The local memory resides in the same location as the global memory, so the local memory has the same latency and memory bandwidth as the global memory. Therefore, the register spilling can lead to severe degradation of the performance.

Both the shared memory and registers are scarce and limited on each SM. Therefore, their usage will impact the actual parallelism supported by each SM. Overuse of these two resources will lower thread parallelism and kill the performance of applications, so we should use them in a proper way.

A modern GPU supports not only kernel level concurrency but also grid-level concurrency. CUDA implements grid-level concurrency through streams. A CUDA stream is a sequence of asynchronous operations executed on a GPU in the order issued by the host. These asynchronous operations include data movement between host and device and kernel launches. A operation will be executed after all preceding operations in the queue are completed. While operations within the same stream maintain a strict ordering,
operations in different streams have no restrictions at all. It is possible to overlap an operation in one stream with operations in other streams, since all the operations queued in streams are asynchronous. Most CUDA applications follow this pattern: copying data from the host to the device, launching a kernel, and moving results back to the host from the device. Hence, one can hide the data movement latency between the host and the device by utilizing multiple CUDA streams. Figure 2.15 demonstrates the benefits of using multiple CUDA streams. $H2D, D2H$ and $K$ stand for the memory copy from host to device, memory copy from device to host, and kernel execution. Since Kepler, a GPU supports up to 32 concurrent CUDA streams.

![Timeline of CUDA Streams](image)

Figure 2.15: Timeline of CUDA Streams

Although all the streams can run simultaneously, the operations in these streams will be queued into a single hardware work queue before Fermi. This single work queue will result in a false dependency because of the task dependencies within each stream. Figure 2.16 illustrates the false dependencies of multiple streams before Fermi architecture.
The false dependency between CUDA streams are solved by introducing multiple work queues since Kepler. Such technology is called Hyper-Q. There are generally 32 work queues on a GPU after Kepler. Each work queue is allocated to one CUDA stream. If more than 32 streams are created, multiple streams will be mapped to one work queue. Figure 2.17 demonstrates the Hyper-Q technology for three streams.
CHAPTER 3

GRAPHIE: GRAPH TRAVERSALS ON A GPU

Most GPU-based graph systems cannot handle large-scale graphs that do not fit in the GPU memory. The ever-increasing graph size demands a scale-up graph system, which can run on a single GPU with optimized memory access efficiency and well-controlled data transfer overhead. However, existing systems either incur redundant data transfers or fail to use shared memory. In this chapter we present Graphie, a system to efficiently traverse large-scale graphs on a single GPU. Graphie stores the vertex attribute data in the GPU memory and stream edge data asynchronously to the GPU for processing. Graphie’s high performance relies on two renaming algorithms. The first algorithm renames the vertices so that the source vertices can be easily loaded to the shared memory to reduce global memory accesses. The second algorithm inserts virtual vertices to the vertex set to rename real vertices, which enables the use of a small boolean array to track active partitions. The boolean array also resides in shared memory and can be updated in constant time. The renaming algorithms do not introduce any extra overhead in the GPU memory or graph storage on disk. Graphie’s runtime overlaps data transfer with kernel execution and reuses transferred data in the GPU memory. The evaluation of Graphie on 7 real-world graphs with up to 1.8 billion edges demonstrates substantial speedups over X-stream, a

---

1Reprinted with permission, from Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU in IEEE International Conference on Parallel Architectures and Compilation Techniques, 2017

†Primary researcher and author

‡Department of Computer Science, Colorado School of Mines, 1500 Illinois Street, Golden, Colorado 80401, USA.

‡Corresponding author. Direct correspondence to bwu@mines.edu.

³Associate Professor Department of Computer Science, Colorado School of Mines, 1500 Illinois Street, Golden, Colorado 80401, USA.
state-of-the-art edge-centric graph processing framework on the CPU, and GraphReduce, an out-of-memory graph processing systems on GPUs.

3.1 Introduction

Graphs are used in various domains, such as machine learning, social networking, and bioinformatics, thanks to the flexible modeling capability. With the ever-increasing graph sizes, it becomes critical to improve the performance of graph processing, because many applications do not tolerate latency well. The problem is, however, challenging due to the well-known random access problem and dramatic behavioral changes across phases of the same application and across inputs.

To accelerate large-scale graph analytics, researchers have proposed many distributed graph processing systems [40, 19, 25]. PowerGraph [18] considers the power-law distribution of vertex degrees and implements a vertex-cut partitioning method to reduce inter-machine communication and improve load balance. PowerLyra [10] further improves the performance by selectively applying vertex-cut and edge-cut approaches that match the characteristics of different parts of the graph. Although those distributed graph systems provide impressive performance, users may still prefer a single-machine based graph system, which is easy to manage and understand [35]. GraphChi [35] is the first graph system that can process large-scale graphs with decent performance on a single machine. X-stream [59] proposes the edge-centric processing model which sequentializes access to edge data. Galois [50] implements a data-centric model and avoids redundant computations that may occur in other systems.

With the increasing popularity of GPU computing, many systems are equipped with GPUs to accelerate various types of tasks. Scaling up graph processing on a single GPU also attracted substantial attention [47, 46, 30, 37]. CuSha [32] implements G-Shard, a similar data structure as used in GraphChi, which optimizes memory coalescing. Gunrock [73] provides a set of high-level primitives, which demonstrate an order of magnitude speedup over PowerGraph. Unfortunately, neither CuSha nor Gunrock can
process graphs that do not fit in the GPU memory. However, many real-world graphs have billions of edges, and the size of the edge data alone (e.g., 11 GB for the Twitter graph used in this work) can be easily larger than the limited GPU memory size (e.g., 6GB for the Nvidia Titan GPU).

In this chapter, we focus on large-scale graph traversals, such as breadth-first search (BFS) and connected components (CC), most existing GPU-based graph systems cannot handle. We face three major challenges. First, a traversal touches a large amount of data but performs little computation. For example, prior work shows that the ratio between data transfer time and kernel execution time on real-world graphs can be up to 2 [33], indicating that data transfer may dominate the execution. Second, the random access problem leads to poor GPU memory efficiency, and meanwhile makes it hard to leverage shared memory. Third, the frontier (the set of active vertices) of a graph traversal changes along the execution depending on the topology of the graph.

GraphReduce [61] and GTS [33] are two existing GPU-based graph systems that claim to be able to process out-of-memory graphs. But neither of them well addresses all the three challenges. For example, GraphReduce heavily optimizes for GPU memory access efficiency. It uses the Compressed Sparse Column (CSC) format for the gather phase and Compressed Sparse Row (CSR) format for the scatter phase. Transferring both CSC and CSR data contains substantial redundancy, which worsens the GPU memory pressure and lengthens data transfer time. GTS can adapt to the dynamic frontiers and avoid redundant data transfers. In addition, its slotted page format helps improve load balance and memory coalescing. But GTS fails to exploit shared memory, and its graph representation is rarely seen in the graph processing field.

In this chapter, we present Graphie, the first GPU-based graph system that addresses all the three challenges of large-scale graph traversal. It overcomes the GPU memory capacity limitation and can efficiently process graphs with billions of edges. Graphie uses one of the most popular graph formats, edge list, and divides it into partitions. It keeps the vertex attribute data in the GPU memory, and streams the edge partitions to the
GPU. Unlike current systems (e.g., GraphReduce), Graphie does not introduce any redundancy besides the edge data. Its optimized performance comes from one key idea: vertex renaming. The renaming has two rounds powered by two algorithms. Once the first-round renaming is done, Graphie allows efficient use of shared memory to accelerate vertex attribute data accesses, as well as improving memory coalescing. After the second-round renaming, Graphie can use a small boolean array to keep track of the partitions that contain active vertices as source vertices and hence should be transferred to the GPU. Graphie stores the boolean array in shared memory, and updates its elements in constant time, which is infeasible without renaming. Graphie hides data transfer overhead through asynchronous streaming and avoids redundant data transfers by reusing edge partitions already resident in the GPU memory. These techniques combined together make Graphie substantially outperform X-stream (up to 98X performance improvement), a state-of-the-art edge-centric graph processing framework on the CPU. Since Graphie is the only open-sourced GPU-based graph system to process out-of-memory graphs 1, we cannot directly compare the performance with GraphReduce [61], which is a similar system but unfortunately not released to public. However, although our used GPU is just slightly more powerful than the GPU used in the GraphReduce work (details in Section 4.7), on the same set of non-trivial graphs used by GraphReduce the results of Graphie demonstrate up to 179X speedup over the results reported for GraphReduce.

We make the the following contributions in this chapter:

1. We propose two renaming algorithms to improve large-scale graph traversal’s performance on GPUs. The first algorithm enables efficient use of shared memory for accessing vertex attribute data. The second enables the use of a small boolean array in shared memory to track the active partitions that should be transferred to the GPU. Neither algorithm introduces any space overhead in the GPU memory or in the graph storage on disk.

2. We propose an asynchronous edge streaming runtime, which hides data transfer

---

1 We elide the link to the source code because of the anonymous reviewing requirement.
overhead and efficiently reuses transferred data across super steps.

3. We integrate the renaming algorithms and the runtime into a GPU-based graph system named Graphie, which supports expressive graph algorithm programming and the traversal of graphs with billions of edges.

4. We evaluate Graphie on 7 real-world and synthetic graphs used in various studies. The results show that Graphie produces up to 98X speedup over X-stream. When processing small graphs, Graphie’s performance is comparable to CuSha, a high-performance GPU-based system to process in-memory graphs.

3.2 Background and Motivation

This section first provides the background of the vertex-centric and edge-centric graph processing models, and explains the reason for Graphie to choose the edge-centric model. It then presents the high-level framework to process out-of-memory graphs on GPUs. It motivates the work by describing the performance issues an optimizing graph system must address.

3.2.1 Graph processing models and data organization

There exist many models for single-machine large-scale graph processing, such as vertex-centric [35], edge-centric [59], data-centric [50], path-centric [76], and matrix-based [64] models. We limit the discussion to the vertex-centric model, represented by GraphChi [35], and the edge-centric model, represented by X-Stream [59], because they are extensively studied and implemented in many systems. Algorithm 2 shows the high-level workflow of the vertex-centric mode, which divides the vertices into vertex partitions. During the processing of each vertex partition, the accesses to the vertices have good spatial locality, while the accesses to the in_edge and out_edge are random. To the contrary, the edge-centric model, shown in Algorithm 3, divides the edges into partitions and enables sequential accesses to edges. However, the accesses to vertices are random as a downside. Because the number of edges is typically much larger than the number of vertices, the edge-centric model outperforms its counterpart demonstrated by multiple systems [59, 58, 33].
Algorithm 2: Vertex-centric Model

Data: Graph

Result: Vertex Value

1 for v ∈ Vertex_partition do
2    if v.active then
3        for e ∈ v.in_edges do
4            Process_in_edge(e)
5        end
6        for e ∈ v.out_edges do
7            Process_out_edge(e)
8    end
9 end

Algorithm 3: Edge-centric Model

Data: Graph

Result: Vertex Value

1 for e ∈ edge_partition do
2    if e.src.active then
3        // Read e.src
4        send_update_over_edge(e)
5        // Write e.dst
6        apply_update(e.dst)
7 end
8 end

3.2.2 Out-of-memory graph traversals on GPUs

GPUs have been successfully used for in-memory graph traversals [46, 53, 73]. The graph data only need to be copied at the beginning of the processing, whose overhead is
amortized to the many phases of traversals. Once the whole graph data is readily available in the GPU memory, systems such like CuSha [32] or GunRock [73], can provide up to two orders of magnitude performance improvement over state-of-the-art CPU-based graph systems. However, GPUs have limited main memory. A high-end GPU, such as Nvidia Titan Z, is only equipped with 6GB memory, while many real-world graphs have billions of edges and are hence too large to fit in the GPU memory. Table 3.1 shows 7 graphs used in this chapter, which are used by many other studies [61, 33, 59]. Suppose an edge needs 8 bytes, 4 bytes for the source vertex ID and 4 bytes of the destination vertex ID. The graph Friendster’s topology data (i.e., edges) alone needs 14GB memory space. Since the execution also needs to store the vertex attributes and possibly edge weights, the actual memory requirement can be significantly larger.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>cage15[1]</td>
<td>5.1M</td>
<td>25.6M</td>
</tr>
<tr>
<td>kron_g500_logn21[57]</td>
<td>2.1M</td>
<td>100.7M</td>
</tr>
<tr>
<td>nlpkkt160[60]</td>
<td>8.3M</td>
<td>110.6M</td>
</tr>
<tr>
<td>orkut[75]</td>
<td>3.1M</td>
<td>117.2M</td>
</tr>
<tr>
<td>uk-2002[2]</td>
<td>18.5M</td>
<td>298.1M</td>
</tr>
<tr>
<td>friendster[75]</td>
<td>65.6M</td>
<td>1,806.1M</td>
</tr>
<tr>
<td>twitter[34]</td>
<td>61.6M</td>
<td>1,468.4M</td>
</tr>
</tbody>
</table>

Algorithm 4 shows the basic workflow to process out-of-memory graphs on a GPU. The function \textit{ProcessGraphOnGPU} runs on the CPU and takes a graph \( G \) stored in the CPU memory as the input. It initializes the vertex attribute array \( VA_{CPU} \) and copies it to the GPU memory. The assumption is that the GPU memory is large enough to hold the vertex attribute array, which is true for most real-world graphs [32, 33, 61]. The graph’s edge data are divided into partitions (i.e., \( G.edge\_partitions \)). The size of the partitions is chosen such that a partition can reside in the GPU memory together with the vertex attribute array \( VA_{GPU} \). Each iteration of the while loop represents a super step, whose finishing point implicitly indicates a global synchronization. The loop body transfers the edge partitions one by one and invokes a kernel to process the transferred edge partitions.
to update the attribute array. Note that the edge partitions are read-only and can be
safely overwritten after being processed. The kernel function \textit{PartitionKernel} launches as
many threads as the number of edges in the partition. One thread corresponds to one edge
and calls the update device function if the edge’s source vertex is active. The Update
function may update the attribute data of the destination vertex. If an update happens,
we say the destination vertex is activated. Both GTS and GraphReduce implement a
similar workflow.

Algorithm 4: Basic workflow to process out-of-memory graphs on the GPU.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Function} \textit{ProcessGraphOnGpu}(G):
\State \hspace{1em} \text{\textbf{Data:} Input Graph } G
\State \hspace{1em} \text{\textbf{Result:} Vertex Value}
\State \hspace{1em} \text{\textbf{Function} \textit{ProcessGraphOnGpu}(G):}
\State \hspace{2em} \text{\textbf{Variable} \textit{VA}_{CPU} ← \text{initialize vertex attribute}(G)}
\State \hspace{2em} \text{\textbf{while} not finished \textbf{do}}
\State \hspace{3em} \text{\textbf{Transfer data} \textit{VA}_{CPU}, \textit{VA}_{GPU}, \text{CpuToGpu}}
\State \hspace{3em} \text{\textbf{foreach} } EP_{CPU} \text{ \textbf{in} } G.edge partitions \textbf{do}
\State \hspace{4em} \text{\textbf{Transfer data} \textit{EP}_{CPU}, \textit{EP}_{GPU}, \text{CPUToGPU}}
\State \hspace{4em} \text{\textbf{PartitionKernel} } \textit{EP}_{GPU}, \textit{VA}_{GPU}, \cdots \textbf{end}
\State \hspace{2em} \text{\textbf{transfer data} \textit{VA}_{CPU}, \textit{VA}_{GPU}, \text{GPUToCPU}}
\State \hspace{1em} \text{\textbf{return} } \textit{VA}_{CPU}
\State \hspace{1em} \text{\textbf{Function} \textit{PartitionKernel}(edge partition):}
\State \hspace{2em} \text{\textbf{Variable} \textit{tid} ← get thread id()}
\State \hspace{2em} \text{\textbf{Variable} \textit{e} ← edge partition[tid]}
\State \hspace{2em} \text{\textbf{if} \textit{e.src} is active \textbf{then}}
\State \hspace{3em} \text{\textbf{Up} \textit{VA}[e.dst], \textit{VA}[e.src]}
\State \hspace{2em} \text{\textbf{end}}
\State \hspace{1em} \text{\textbf{return}}
\end{algorithmic}
\end{algorithm}

Figure 3.1 shows an example graph and its edge partitions. The graph has eight
vertices and 16 edges. Each edge partition has four edges. We do not assume the edges are
sorted, because the input graphs may not have been pre-processed. Suppose vertex 5 is the
root node for a BFS traversal. When the execution starts, it is the only active vertex. We
further suppose each kernel invocation launches one single thread block of four threads to
process the transferred edge partition. When processing the third edge partition, the first
thread would activate vertex 1. The GPU can successfully process this graph if the memory is large enough to hold the vertex attribute data (eight variables) and one edge partition (four edges). In the remainder of the chapter, whenever this example is used, we have the same assumption about the thread block size and the partition size. We also assume the thread block can only load four vertices to the shared memory, which can be viewed as software-controlled cache.

Algorithm 4 shows that to improve the performance we should reduce the data transfer overhead (line 6) and/or improve the kernel’s performance (line 7). We next present the challenges of achieving these goals using the example shown in Figure 3.1.

**Issue 1: dynamic frontiers.** Graph traversals typically have complicated behaviors depending on the algorithm and graph topology. Specifically, the number of active vertices (i.e., the frontier) and their distribution in the vertex set may vary dramatically across super steps. For example, BFS starts with one active vertex (i.e., the root vertex) and in each super step activates a new set of vertices which are just discovered in this step. Connected component, on the other hand, has all vertices as active vertices at the beginning of the execution. The number of active vertices decreases towards the end of the execution. In the context of graph traversals on the GPU, the transfer of an edge partition is redundant if none of its edges are out-edges of active vertices. For instance, 3.1 shows that in the second super step of a BFS run with vertex 6 as the root, the second edge partition contains all the out-edges of the activated vertex (i.e., vertex 1), and hence is the only one that should be transferred to the GPU for optimized performance.

Current graph systems that support out-of-memory graph processing on GPUs solve this problem by keeping track of the active vertices via a boolean array. The size of the array is the number of vertices in the input graph, because any vertex may be active for the next super step. This approach has three drawbacks. First, despite contributing nothing to the real computation, this meta array stays in the GPU memory and incurs non-trivial space overhead (e.g., 262MB for friendster). Second, the array needs to be copied back to the CPU at the end of each super step, causing time overhead on the critical path. Third,
the array is too large to fit in GPU’s shared memory. As such, every update causes one extra GPU main memory access to update the corresponding boolean variable.

**Figure 3.1:** An example graph and its partitioned edge list.

<table>
<thead>
<tr>
<th>SRC:</th>
<th>0 0 0 6</th>
<th>1 1 7 6</th>
<th>5 2 6 7</th>
<th>7 0 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DST:</td>
<td>1 2 3 2</td>
<td>4 0 2 0</td>
<td>1 3 3 6</td>
<td>4 6 4 5</td>
</tr>
</tbody>
</table>

**Issue 2: memory access inefficiency.** GPU kernel’s performance highly relies on memory access efficiency. There are two major ways to improve the efficiency. First, if the threads running on the same SIMD unit access nearby memory locations, the memory accesses may be coalesced to reduce the number of memory transactions. Second, if the threads of the same thread block repeatedly access the same memory location, the data element at that location should be fetched to shared memory for those threads to quickly access. A naive implementation of Algorithm 4 fails to exploit either memory coalescing or shared memory. The processing of the partitions incurs random vertex accesses and hence excessive uncoalesced memory transactions. The randomness also complicates the use of shared memory, which needs heavyweight pre-processing to figure out the set of accessed
vertices, easily canceling the benefit.

In the next two sections, we propose several techniques to address these two issues, followed by the presentation of the Graphie framework that integrates these techniques for efficient large-scale graph traversal.

3.3 Optimizing Kernel Execution through Vertex Renaming

This section discusses the inadequacy of existing solutions to the performance issues described in last section. It presents two vertex renaming algorithms to improve memory access efficiency and efficiently find out the active partitions that should be transferred. The section explains why the renaming process does not introduce any space overhead in the GPU memory.

3.3.1 Improving GPU memory access efficiency

A naive solution to improving the memory access efficiency problem is to sort the edges by source vertex ID. Figure 3.2a shows the sorted edge list of the example in 3.1. We note that the sorting improves the performance of the accesses to the source vertices because of enhanced locality but with the accesses to the destination vertices remaining random.

It may seem after the sorting, the distinct source vertices can be loaded into the shared memory to reduce main memory accesses. However, although the number of distinct vertices is up to the number of edges in the partition, the gap between the first source vertex ID and the last source vertex ID can be larger than the number of vertices that can be loaded to shared memory. The reason is that many vertices whose IDs are in between do not have out-going edges. As Figure 3.2a shows, the second partition has three distinct vertices, but the gap is 4. Recall that we assume the shared memory used by one thread block can hold up to 4 vertices, so we cannot load 5 vertices (2–6) to the shared memory. Therefore, even with the sorted edges, it still requires a non-trivial pre-processing phase to figure out the distinct source vertices (i.e., vertices 1, 2 and 5 in the example).
(a) Sorted edges by source

(b) The first-round renaming
To address these problems, we propose a renaming technique which not only improves memory coalescing but also makes using shared memory straightforward. The renaming process happens after the edges are sorted by source vertex. Algorithm 5 shows how the technique works through two functions: RenameForMemory to rename the vertices and PartitionKernelV2 to demonstrate the convenient use of shared memory. The idea of RenameForMemory is to rename the vertices with out-going edges by contiguous smaller IDs than those of the vertices with no out-going edges. It first scans the edges to compute the out-going degree for each vertex (line 2). It then uses an array new_to_old to compute the new IDs for the vertices. After the first for loop, the vertex of the ID given by new_to_old[i] should have the new ID i. To quickly access the new ID given the old ID, we use an array old_to_new, whose ith element is the new ID of the ith vertex in the original graph (line 10). Finally, the out-edges of each vertex are sorted by destination (line 16–18), which improves the spatial locality of accessing destination vertices, thus improving memory coalescing.
After the edges are renamed and reordered, *PartitionKernelV2* shows the convenient use of shared memory in the kernel. Since the IDs of the source vertices of the edges are contiguous, we easily calculate the number of distinct source vertices based on the source vertex IDs of the first edge and the last edge (line 26). We only load the attribute variables of the distinct source vertices to shared memory (lines 28 and 29) followed by a thread block-level barrier to avoid data races. The Update function *Up* accesses the attribute variables of the source vertices in the shared memory (line 33), which may significantly reduce the number of global memory accesses because those source vertices may have many out-going edges.

Once the processing on the GPU finishes, we transfer the vertex attribute array (*VA_GPU*) back to the CPU to store in the array *VA_CPU*. However, because the vertices are renamed, we need to map the updated attribute data to the corresponding vertices. The problem can be easily solved, because we maintain the mapping in the array *new_to_old* obtained in *RenameForMemory*. The *i*th attribute variable *VA_CPU[\[i\]*] should belong to the vertex of ID *new_to_old[\[i\]*] in the original graph.

Figure 3.2b shows the renamed and reordered graph data of the example graph after being processed by Algorithm 5. Vertex 3 and 4 in the original graph have new IDs 6 and 7 (i.e., the largest IDs), respectively, because they do not have out-going edges. Correspondingly, the IDs of vertex 5–7 in the original graph are reduced by 2. Observe that the source vertices of the edges are contiguous and that the out-going edges of the same source vertex are sorted by destination vertex ID. Unlike Figure 3.2a, we can now easily compute the number of distinct source vertices of the second partition thanks to the source ID contiguity.

*RenameForMemory* runs online or offline on the CPU to pre-process the edge data. It requires the edges to be sorted by source ID, but most systems demand a similar sorting process (e.g., GraphChi and GTS). The rationale of the pre-processing is that it is done just once and the cost can be amortized to many runs with different inputs such as different roots for BFS or with different algorithms on the same graph. In
RenameForMemory, each state before the final sorting (line 13) takes linear time in terms of the number of vertices or the number of edges. The sorting stage takes \(O(N \times \Delta \log(\Delta))\), where \(\Delta\) is the maximum outdegree. Therefore, RenameForMemory’s worst complexity is \(O(M + N \times \Delta \log(\Delta))\), where \(M\) is the number of edges. If we assume \(\Delta\) to be the average degree multiplied by a constant, the complexity can be estimated as \(O(M + N \times (M/N) \times \log(M/N)) = O(M \log(M/N))\), which is less than what sorting takes (i.e., \(O(M \log(M))\)). Moreover, Algorithm 5 does not incur any extra space overhead in the GPU or extra storage overhead in the disk.

3.3.2 Efficiently Activating Partitions

Recall that graph traversals have dynamic frontiers, and only the edge partitions that contain one or more vertices in the dynamic frontier should be transferred to the GPU to save data transfer time. As discussed in Section 3.2.2, both GTS and GraphReduce use an array of size \(N\) (i.e., the number of vertices) to record which vertices are in the frontier for the next super step, which burdens the GPU memory and increases data transfer cost. Worse, the array is too large to benefit from shared memory. To address this problem, the ideal solution is to have a small boolean array of size \(P\) (also called a tag array), the number of partitions, on the GPU to track the partitions that contain active vertices as source vertices. For example, for the graph shown in Figure 3.2b, we only need a boolean array of size 4 (instead of 16 in existing systems). If only vertex 2 is activated in the current super step, the boolean array should be \{False, True, False, False\}. After processing this array, the CPU knows that only partition 2 should be transferred to the GPU in the next super step, as it contains all the out-going edges of the vertex 2. However, computing which partition contains the activated vertex involves searching and needs \(O(\log P)\) time. Since the overhead occurs every time a vertex is updated, this approach may perform worse than the existing approach of maintaining a large boolean array.
Algorithm 5: Renaming vertices to improve memory access efficiency.

Data: Input Graph $G$

Result: Vertex Value

// $G$ is the input graph
// $N$ is the number of vertices

1 Function RenameForMemory($G$):
   2 $out\_degrees \leftarrow $ ComputeOutDegrees($G$)
   3 $new\_to\_old \leftarrow \{0, 1, \ldots, N - 1\}$
   4 foreach $id$ in $new\_to\_old$ do
      5      if $out\_degrees[id] = 0$ then
      6         move $id$ to the end of $new\_to\_old$
      7   end
   8 end
   9 for $i \leftarrow 0$ to $N - 1$ do
      10      $old\_to\_new[new\_to\_old[i]] \leftarrow i$
   11 end
   12 forall the $e$ in $G$.edges do
      13      $e$.src $\leftarrow old\_to\_new[e$.src$]$
      14      $e$.dst $\leftarrow old\_to\_new[e$.dst$]$
   15 end
   16 forall the $v$ in $G$.vertices do
      17      if $out\_degrees[v$.id$] \neq 0$ then
      18         Sort $v$'s outgoing edges by destination
      19   end
   20 end
   21 return

// $BS$ is the thread block size
// $SVA$ is an array in the shared memory

23 Function PartitionKernelV2($EP$, $VA$):
   24 $tid \leftarrow get\_thread\_id()$
   25 $start\_vertex \leftarrow EP[0]$.src
   26 $num\_distinct\_srcs \leftarrow EP[BS - 1]$.src $-$ $EP[0]$.src
   27 $e \leftarrow edge\_partition[tid]$
      // Load attributes of distinct vertices to shared memory
   28 if $tid < num\_distinct\_srcs$ then
      29      $SAV[tid] \leftarrow VA[start\_vertex + tid]$
   30 end
   31 Barrier() // synchronize threads to avoid data race
   32 if $e$.src is active then
      33      Up($VA[e$.dst$], SVA[e$.src$ $-$ $start\_vertex$])
   34 end
   35 return
We propose a technique to further rename the vertices based on the renamed graph produced by Algorithm 5. Algorithm 6 shows the process of reducing the cost of figuring out the activated partition to one single division operation. The essential idea is to insert virtual vertices, which do not need storage, to the source vertex sets of partitions, so that the IDs of distinct source vertices of each partition fall into ranges of the same size. Lines 8 and 9 calculate for each partition the difference between its number of distinct source vertices and the maximum number of distinct vertices across partitions.

$num_{\text{virtual\_vertices}}$ of size $P$ maintains the numbers of virtual vertices that should be inserted. $pre_{\text{sum}}$, computed via an inclusive scan on $num_{\text{virtual\_vertices}}$, stores the total number of inserted virtual vertices before each edge partition. With 0 appended at the front, $pre_{\text{sum}}[P]$ is now the total number of inserted virtual vertices. The vertex IDs of the source vertices of each partition $i$ should be increased by $pre_{\text{sum}}[i]$ to reflect the number of inserted virtual vertices before them. Similarly, the IDs of the vertices without out-going edges should be increased by $pre_{\text{sum}}[P]$. The renaming of the IDs in edge data is the same as in Algorithm 5.

We next show how the renaming enables constant time update of the boolean array $activated$ in the new kernel function $PartitionKernelV3$. Recall that the size of $activated$ is equal to $P$, the number of partitions, and hence the array is usually small enough to be stored in shared memory.

One key difference from $PartitionKernelV2$ in Algorithm 5 is that when the attribute array is accessed, the index should be decreased by an offset (lines 37 and 40). The offset equals the number of inserted virtual vertices before the partition that contains the vertex as a source vertex. Lastly, if the update function returns true, meaning that the destination vertex ($e.dst$) is updated, and the updated vertex has out-going edges, the partition that contains it as a source vertex should be activated. The ID of that partition can be easily calculated as $e.dst/SZ_{\text{MAX}}$, a tremendous improvement over naive searching.
Algorithm 6: Renaming vertices to efficiently activate partitions.

// G is the graph produced by Algorithm 5
// N is the number of vertices
// P is the number of partitions
// num_virtual_vertices is an array of size P initialized to all 0’s
// new_ids is initialized as \{0,1,\ldots,N-1\}

Function RenameForActivePartitions(G):

//SZ stores the number of distinct source vertices for all partitions
scan edges to compute \(SZ\)
compute \(SZ_{\text{max}}\) //the maximum number in \(SZ\)
for \(i \leftarrow 0\) to \(P - 1\) do
  num_virtual_vertices \(\leftarrow SZ_{\text{max}} - SZ[i]\)
end
pre_sum \(\leftarrow \text{InclusiveScan}(\text{num\_virtual\_vertices})\)
append 0 at the front of \(\text{pre\_sum}\)
for \(i \leftarrow 0\) to \(P - 1\) do
  foreach \(v \in EP[i].\text{distinct\_source\_vertices}\) do
    new_ids[v.ID] \(\leftarrow new_ids[v.ID] + \text{pre\_sum}[i]\)
  end
end
foreach vertex \(v\) without out-going edges do
  new_ids[v.ID] \(\leftarrow new_ids[v.ID] + \text{pre\_sum}[P]\)
end
foreach \(e\) in G.edges do
  e.src \(\leftarrow new_ids[e.\text{src}]\); e.dst \(\leftarrow new_ids[e.\text{dst}]\)
end
return

// activated is an array of size P in the shared memory initialized to all False
// OV is the number of vertices that have out-going edges

Function PartitionKernelV3(EP, VA):

tid \(\leftarrow \text{get\_thread\_id}()\)
start_vertex \(\leftarrow EP[0].\text{src}\)
num_distinct_srcs \(\leftarrow EP[BS - 1].\text{src} - EP[0].\text{src}\)
e \(\leftarrow \text{edge\_partition}[\text{tid}]\)
if \(e.\text{src} < OV\) then
  src_offset \(\leftarrow \text{pre\_sum}[e.\text{src}/SZ_{\text{max}}]\)
end
else
  src_offset \(\leftarrow \text{pre\_sum}[P]\)
end
if $e.\text{dst} < \text{OV}$ then
\[ \text{dst.offset} \leftarrow \text{pre.sum}[e.\text{dst}/\text{SZ.max}] \]
end
else
\[ \text{dst.offset} \leftarrow \text{pre.sum}[P] \]
end
if $\text{tid} < \text{num\_distinct\_srcs}$ then
\[ \text{SAV[tid]} = \text{VA[start.vertex + tid - src.offset]} \]
end
\text{Barrier}() //synchronize threads to avoid data race
if $e.\text{src}$ is active then
\[ \text{if Update(VA[e.\text{dst} - \text{dst.offset}], SV A[e.\text{src} - \text{start.vertex}]) and} \]
\[ e.\text{dst} < \text{OV} \text{ then} \]
\[ \text{activated[e.\text{dst}/\text{SZ.max}] } \leftarrow \text{True} \]
end
return;

Figure 3.2c shows the renamed graph by Algorithm 6 based on Figure 3.2b. Partition 2 has 3 distinct source vertices, the largest among all the partitions. All the other partitions only have 1 distinct source vertex. Hence, we insert 2 virtual vertices in each of the source vertex set of partition 1, 3, and 4. Suppose a thread processes edge $(5, 3)$ and needs to update vertex 3. It writes vertex 3’s new value to $\text{VA[3 – 2]}$ (i.e., $\text{VA[1]}$). The ID of the partition that contains 3 is easily computed as $3/3 = 1$. The boolean variable $\text{activated[1]}$ is then assigned to True.

We stress that although Algorithm 6 may insert a large number of virtual vertices, it, like Algorithm 5, does not introduce extra space overhead in the GPU memory, because we only need $N$ elements in the attribute array. All the steps of $\text{RenameForActivePartitions}$, including the nested loop (lines 10–12), can be implemented in linear time. Thus, the time complexity is $O(M)$.

3.4 Minimizing Data Transfer Overhead through Asynchronous Streaming

This section presents the techniques used by Graphie to reduce data transfer overhead through asynchronous edge streaming.
Graph traversals are memory-intensive with very low arithmetic intensity. As such, the data transfer of a partition may take longer than its processing on the GPU. Fortunately, modern GPUs support parallel command queues (e.g. Hyper-Q in Nvidia GPUs [52]), which allow overlapping between kernel execution and data transfer. Graphie leverages this capability to hide edge partition transfer overhead as shown in Figure 3.3. After initializing the vertex attribute array (VA), Graphie divides the remaining GPU memory into $K$ partition buffers, where $K$ is the number of streams, unless the user explicitly specifies the partition size. Graphie takes turns to use the streams to transfer the edge partitions to the GPU. In each stream, a kernel invocation commands always follows a data transfer command to process the transferred partition. The commands sent to the same command queue are executed sequentially. Current Nvidia GPUs support up to 32 command queues. If more streams are used, some streams will be serialized to use the same command queue. Hence, Graphie uses 32 streams by default unless specified otherwise.

Section 3.3.2 described the renaming technique to efficiently identify the activated partitions to process in the next super step. Graphie’s runtime makes sure that only activated partitions are transferred. While this optimization greatly reduces the data transfer overhead when the number of activated partitions is small, redundant transfer may still occur if the activated partition is already in the GPU memory. Suppose the average
number of activated partitions in each super step is $A$. On average, the percentage of redundant partition transfers can be estimated as $\frac{A \times K}{P}$, which can be non-trivial if the graph size is not dramatically larger than the GPU memory size.

Because of the FIFO property of the command queue, it is obvious that the last processed partition by each queue can be reused in the next super step. However, to reuse those partitions, it is critical to not overwrite them before processing them. Graphie solves this problem by first processing the partitions that are activated and also resident in the GPU memory. For each of such partitions, Graphie inserts only the kernel invocation command to the queue which handled that partition in the last super step.

3.5 Graphie System

3.5.1 Graphie workflow

We integrate all the proposed techniques in the Graphie system, whose workflow is shown in Figure 3.4. The circled numbers represent different steps. Graphie reads edge data in text or binary format (step 1). It stores the edge data in an edge list, which is processed by the renaming engine for renaming and partitioning (step 2). Note that the renaming engine can work online or offline. For each edge partition, Graphie uses one array to store the source vertex IDs and one array to store the destination vertex IDs. Graphie uses one more array to store the weight data if there is any. Such a design choice is to improve memory coalescing, which is used in many other GPU-based systems. The runtime engine reads all edge partitions in the CPU memory (step 3). In each super step, it transfers edge partitions to the GPU and invokes kernels to process the partitions (step 4). At the end of each super step, it copies back the flag array (i.e., activated in Algorithm 6). If any partition is activated, it starts another super step that consists of steps 4 and 5. Once it detects no active partitions, the output vertex attribute array is remapped to the original vertex ids to cope with renaming.
Figure 3.4: The workflow of Graphie.
3.5.2 Programming interface

Graphie provides a generic kernel, which implements the PartitionKernelV3 function in Algorithm 6. It invokes two device functions that the user must implement. The first device function is Initialize VA, which should initialize the vertex attribute array. The implementation is application dependent. For example, for BFS VA[root] should be initialized to 0, while all other elements should be initialized as positive infinity. The second device function is Update, which processes an edge if the source vertex is active, and returns true if the destination vertex of the edge is updated (i.e., activated). For BFS, the destination vertex is updated if and only if its distance (i.e., VA_Gpu[e.dst]) is larger than the distance of the source vertex (i.e., VA_Gpu[e.src] + 1 < VA_Gpu[e.dst]). On the CPU side, the user needs to specify which partitions are active and hence should be processed on the GPU in the first super step. For example, the partition that contains the root vertex should be active for BFS, while all the partitions should be active for CC.

3.5.3 Selecting partition size

As discussed in Section 3.4, Nvidia GPUs support up to 32 parallel command queues. We use GS, VS, AS to denote the size of the GPU memory, the size of the vertex data, and size of the flag array, respectively. Given 32 streams, the partition size can be computed as (GS − VS − AS)/32. However, this partition size does not address the various properties of graphs. For small graphs, the whole graph may fit in one partition. As such, it only uses one stream and does not leverage the concurrency of the command queues. The kernel computation starts after the whole graph is transferred, wasting the opportunity to overlap transfer and compute. It is also possible that a traversal only accesses a subset of the edges. In this case, transferring the whole graph is not necessary. Due to these reasons, Graphie chooses partition size as ES/32, where ES is the edge data size, to fully utilize all the available command queues and avoid unnecessary data transfer. For large graphs, it may be infeasible to support (GS − VS − AS)/32 as the partition size. Recall that during partitioning, Graphie inserts virtual vertices to the source vertex sets.
for renaming, which increases the largest vertex ID. Because Graphie uses 32 bits to store
the vertex ID, a small partition size may cause an integer overflow problem. Therefore,
Graphie chooses the smallest partition that does not overflow the 32-bit integer. Note that
we can simply address the problem by using 64-bit representation for vertex IDs. However,
we then increase the edge data size by 2 times, leading to increased data transfer overhead.
It is our future work to understand the trade-off between the vertex representation and
partition size.

3.6 Experimental Evaluation

This section evaluates the performance of Graphie by comparing it with existing
systems and quantifies the effectiveness of the proposed optimization techniques. Before
presenting the results, we introduce the experiment settings and the methodology for the
experiments.

3.6.1 Experiment setting.

Our system has an Intel Xeon (E7-4830v3, 2.1GHz) 12-core CPU with Hyperthreading
disabled. The main memory of the system is 256GB (16x16GB DDR3 modules at
1866MHz). We use the NVCC compiler version 7.5.17 (g++ version 4.8.4) with O3 to
compile all the programs. The operating system is Ubuntu Linux 16.04 with Linux kernel
version 3.13. The GPU is a NVIDIA Titan Z containing 2 GPU dies each with 6GB of
memory. In all the experiments, we only use one GPU, and hence the device memory is
limited to 6GB. Table 3.2 shows the specification of a single die of the GPU.

<table>
<thead>
<tr>
<th></th>
<th>Titan Z (half)</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU architecture</td>
<td>Kepler (GK110B)</td>
<td>Kepler (GK110)</td>
</tr>
<tr>
<td>Num. of SMX</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>Memory</td>
<td>6GB GDDR5</td>
<td>5GB GDDR5</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>288 GB/S</td>
<td>208 GB/S</td>
</tr>
<tr>
<td>Num. of CUDA cores</td>
<td>2,688</td>
<td>2,496</td>
</tr>
<tr>
<td>Theoretical throughput</td>
<td>4,494 GFLOPS</td>
<td>3,324 GFLOPS</td>
</tr>
</tbody>
</table>

We evaluate the performance of Graphie using three graph traversal algorithms:

- Breadth-First Search (BFS)
• Connected Components (CC)
• Single-Source Shortest Path (SSSP)

The BFS algorithm traverses the vertices of the graph in order to compute unweighted distances of all vertices from a root vertex. In SSSP the weights are considered and the cost of the cheapest path (in terms of the sum of the weights of its constituent edges) from a root to every vertex is returned. For BFS and SSSP, we always select a vertex in the largest connected component as the root. The CC algorithm finds connected subgraphs of maximal size and returns the component id for each vertex.

We experiment with 7 real-world and synthetic graphs as shown in Table 3.1. Cage15 is an undirected graph describing DNA electrophoresis, 15 monomers in polymer. Kron.g500-logn21 (Kron) is a synthetic graph used in the DIMACS competition. Nlpkkt160 (nlpktt) is a graph generated by a symmetric indefinite KKT matrix when solving a 3D PDE-constrained optimization problem. Orkut and Friendster are graphs from online gaming and social networks. Uk-2002 was obtained from a crawl of the .uk domain in 2002. Twitter is a subgraph of the Twitter follower graph.

We compare Graphie with three graph systems: X-stream [59], CuSha [32], and GraphReduce [61]. X-stream is a state-of-the-art edge-centric graph system. Its superior performance over GraphiChi [35], a vertex-centric graph system, is reported in several studies [59, 61]. To make fair comparisons, we allocate enough main memory for X-stream to load the entire graph. We exclude the IO time, and only measure the graph processing time. CuSha is a high performance GPU-based graph system to process in-memory graphs. For both CuSha and Graphie, we measure the elapsed time between the point the first data transfer from the CPU to the GPU starts and the point the final result data transfer from the GPU to the CPU finishes. GraphReduce is an out-of-memory graph processing system on GPUs, which is also based on edge streaming like Graphie. Unfortunately, GraphReduce is not released to public. We can hence only reference to the reported execution times (data transfer + kernel execution) in [61] on the same set of graphs to make rough comparisons. Note that the reported results in [61] are obtained on an Nvidia Tesla K20c
GPU. Table 3.2 shows the specification comparison between K20c and the Titan GPU used for Graphie. Both GPUs are based on the Kepler architecture. The Titan GPU has slightly larger main memory (6GB vs. 5GB) and a larger number of CUDA cores (2,688 vs. 2,496).

### 3.6.2 Overall results

Table 3.3 summarizes the execution times of Graphie and the three compared systems. We ran each algorithm 10 times using Graphie, XStream, and CuSha. The numbers in Table 3.3 are the average. GraphReduce is not open-sourced yet. Thus, we take GraphReduce’s performance of each algorithm from the paper [61]. The standard deviation of each algorithm’s performance is less than 1% of the corresponding average, so they are not shown in this table. N/A for GraphReduce means the corresponding graph is not used in [61]. O.O.M for CuSha means the graph is too large to fit in the GPU memory. We notice that CuSha can only process 4 of the 7 graphs because of the in-memory processing design. It cannot process uk-2002, though the graph’s size (3.3GB) is smaller than the GPU memory. The reason is that CuSha needs to transform the original graph to the G-Shard representation, which incurs non-trivial space overhead. The transformed graph does not fit in the GPU.

Figure 3.6 shows the substantial speedups of Graphie over X-stream. The most significant improvements are for Friendster, which is the largest graph we experiment with. Graphie accelerates CC on the graph by 98X, bringing down the execution time from 1224.5 seconds from X-stream to only 12.5 seconds. On average, Graphie achieves 7.2X, 15.5X, and 20.3X speedups for BFS, SSSP, and CC, respectively. The results demonstrate the power of using GPUs to process large-scale graph traversals and Graphie’s lightweight but efficient design to match the GPU programming model and architecture.

For Orkut, Cage15, and Kron, Graphie outperforms CuSha for 7 out of the 9 runs. The results are impressive because Graphie is designed for large-scale graph traversal, while CuSha is heavily optimized for in-memory graph processing. Graphie benefits from asynchronous edge streaming and its concise graph representation, while CuSha’s G-Shard
format introduces space overhead and makes it hard to use streams. For NLPKKT, CuSha produces superior performance, because the graph has a very large diameter, which has been shown to cause performance problems for the edge-centric model.

![Benefits Breakdown](image)

Figure 3.5: Benefits breakdown.

Table 3.3: Execution times of Graphie and the compared systems.

<table>
<thead>
<tr>
<th>Application</th>
<th>Framework</th>
<th>Runtime for Graph (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cage15</td>
<td>friendster</td>
</tr>
<tr>
<td>bfs</td>
<td>Graphie</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>XStream</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>GraphReduce</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>CuSha</td>
<td>0.45</td>
</tr>
<tr>
<td>cc</td>
<td>Graphie</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>XStream</td>
<td>5.43</td>
</tr>
<tr>
<td></td>
<td>GraphReduce</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>CuSha</td>
<td>0.6</td>
</tr>
<tr>
<td>sssp</td>
<td>Graphie</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>XStream</td>
<td>4.52</td>
</tr>
<tr>
<td></td>
<td>GraphReduce</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>CuSha</td>
<td>0.57</td>
</tr>
</tbody>
</table>
Figure 3.6: Speedup over XStream.
3.6.3 Breakdown of the optimization benefits

To understand the performance contributions from the proposed techniques, we use OPT1 to represent the optimizations (i.e., shared memory use + only transferring active partitions) enabled by Algorithm 6 and OPT2 to represent the optimization to reuse partitions in the GPU memory. Figure 3.5 demonstrates the execution time savings from OPT1 and OPT2 with the naive implementation as the baseline, which does not use shared memory and transfers all partitions in each super step. In all the runs, asynchronous streaming is enabled, and we will analyze its benefit in Section 3.6.4. The results show 22%–27% average execution time reductions from OPT1 for the three algorithms. The largest performance gain is from the CC execution on Twitter, showing 80% reduction of the execution time (i.e., a 5X speedup). But some executions, such as the three runs on NLPKKT, just show trivial performance improvement, because most of the time all the partitions are active. OPT2 also dramatically improves the performance with average execution time reductions between 28% and 38% across the three algorithms. We observe non-trivial reductions for all runs, confirming the importance of reusing partitions in the GPU memory. Working as a synergy, OPT1 and OPT2 reduce the execution time of the naive implementation by an average of 61% for BFS, 62% for SSSP, and 55% for CC across the inputs.

To explain the results we just discussed, we show in Figure 3.7 the number of active partitions and the number of transferred partitions across super steps for 2 out-of-memory graphs and 2 in-memory graphs. The numbers of active partitions may change dramatically because of the dynamic frontier property of graph traversal algorithms. The patterns for BFS and CC are however very different. BFS starts with 1 active partition, which contains the root vertex. The number of active partitions increases because more vertices and hence partitions are activated. The number decreases at the end of the execution because most of the vertices have been processed. Note that the number of active partitions may remain the same across super steps. It does not mean the
corresponding frontiers have the same size. An partition is active even if it only contains one single active source vertex. Therefore, the minimum and the maximum sizes of the frontier to make all partitions active are \( P \) and \( N \), respectively, which demonstrate a huge gap. The CC algorithm starts with all partitions being active, and the number of active partitions decreases along the execution. The decrease can be fast (e.g., for Twitter) or slow (e.g., for Kron) depending on the topology of the graph.

The number of transferred partitions is always equal to or smaller than the number of active partitions. For the latter, Graphie further improves performance by avoiding the transfer of partitions already in the GPU memory. For the two in-memory graphs, Orkut and Kron, the gap between the two curves is large, because the transferred partitions are never overwritten and hence can be reused if needed. For the BFS runs, Graphie does not transfer any more partitions after the 5th super step, because the transferred partitions already contain all the source vertices reachable from the root vertex. The CC runs behave very differently. All the partitions are transferred in the first super step, and thus the whole graph is in the GPU memory for later execution. For the out-of-memory graphs, Friendster and Twitter, the gap between the two curves is smaller, because the GPU memory is not large enough to hold all the partitions and some partitions may be transferred multiple times.

### 3.6.4 Results on asynchronous streaming

Figure 3.8 shows the performance benefit from asynchronous edge streaming. We observe that for all the 4 graphs and 3 algorithms, using a larger number of streams improves performance in most cases. Friendster can only leverage 16 streams, because Graphie has to use a large enough partition size to avoid the integer overflow problem discussed in Section 3.5.3. For the other three graphs, Graphie produces more than 2X performance improvement by using 32 streams. The speedup is much worse than linear, because the streams contend to use the PCIe bus and the same set of GPU cores.
3.6.5 Overhead of the renaming processes

As pointed out in Section 3.3, the renaming process only needs the input graph rather than any runtime parameters (e.g., root vertex ID for BFS) and hence can be performed offline. Table 3.4 shows the overhead of the two rounds of renaming in seconds. For all graphs, the first-round renaming is more expensive than the second-round renaming, which aligns well with the analysis in Section 3.3. We note that the overall overhead from renaming for the two out-of-memory graphs (i.e., Friendster and Twitter) is oftentimes negligible compared to the execution time of X-Stream.

Table 3.4: The overhead of renaming in seconds.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Round 1</th>
<th>Round 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cage15</td>
<td>1.28</td>
<td>0.95</td>
</tr>
<tr>
<td>Kron</td>
<td>6.51</td>
<td>1.63</td>
</tr>
<tr>
<td>Nlpkkt</td>
<td>2.95</td>
<td>2.35</td>
</tr>
<tr>
<td>Orkut</td>
<td>2.3</td>
<td>4.26</td>
</tr>
<tr>
<td>Uk-2002</td>
<td>6.02</td>
<td>2.67</td>
</tr>
<tr>
<td>Friendster</td>
<td>4.48</td>
<td>2.1</td>
</tr>
<tr>
<td>Twitter</td>
<td>3.99</td>
<td>1.35</td>
</tr>
</tbody>
</table>

3.7 Related Work

To handle large-scale graphs, researchers have designed many distributed graph processing frameworks [18, 39, 65, 10, 40, 19], most requiring the entire graph data, edges and vertex, to reside in main memory during execution. But as several studies have shown [35, 59, 64, 50, 63, 79] a single-machine based system can dramatically reduce management overhead while still providing decent performance. Graphie’s design philosophy aligns well with those studies.

Graph processing on GPUs has also been extensively studied from various aspects, including synchronization trade-off [29], data-driven models [46], dynamic graphs [47], graph optimizing compilers [53, 30], and efficient primitives [73]. All those studies assume the input graph fits in the GPU memory, and the research focus is on reducing synchronization overhead or reducing control and memory divergence. Some studies use multiple GPUs to accelerate graph processing. To name a few, Ben-Nun et al. [6] proposed
GRoute, which supports efficient asynchronous multi-GPU programming to handle irregularity in graph processing. Liu et al. [38] dramatically improved concurrent BFS on up to 112 GPUs. Khorasani et al. [31] improved inter-GPU communication compared with Medusa [80] and TOTEM [17].

Merrill and others [42] first demonstrated that GPU-based graph traversals can perform substantially better than the CPU-based counterparts. The major idea is to use pre-fix sum to efficiently manage fine-grained tasks. Our work uses pre-fix sum to track the mapping between renamed vertices and their attribute data in the GPU memory. Moreover, their work, like CuSha [32], assumes the graph fits in the GPU. Liu and Huang further improved BFS’s performance of in-memory graphs on GPUs through a set of techniques for load balancing and direction optimization. It is unclear whether the proposed techniques work well for out-of-memory graphs on a single GPU.

GraphReduce [61] can process out-of-memory graphs on a single GPU. It optimizes memory coalescing through using two different formats, the benefit of which can be easily cancelled by the redundant data transfers. We show in this chapter that Graphie can directly work on edge lists and its renaming and reordering techniques do not introduce any extra space overhead. Further, GraphReduce does not reuse the already transferred data in the GPU when processing large-scale graphs. GTS [33] can also process out-of-memory graphs on GPUs. It leverages slotted page format, which is not popular in the graph processing area. GTS does not use shared memory for accessing vertex data or keeping track of updated vertices. Graphie efficiently tracks the active partitions using shared memory, which involves negligible transfer overhead for the meta flag array.

Researchers have applied different data reorganization techniques to improve the performance of irregular applications for SIMD-based architectures. Wu et al. [74] studied the complexity of data reorganization for optimized GPU memory accesses and proposed several algorithms to strike different trade-offs. Fauzia et al. [15] implemented a tool to automatically characterize uncoalesced memory accesses and transform the data to reduce the degree of divergence. Ren et al. [55] reorganized the tree data structure to improve the
performance of CPU vectorization. Jiang et al. [28] studied the reuse of reorganized data for dynamic irregular applications.

3.8 Conclusion

In this chapter, we presented Graphie, a GPU-based graph system to perform large-scale graph traversals. Graphie leverages asynchronous edge streaming to stream edge partitions to the GPU to hide data transfer overhead. Different from existing systems with a similar architecture, Graphie improves performance of graph traversal through a novel renaming technique. The renaming process consists of two rounds to enable the convenient use of shared memory and efficient activation of edge partitions, which does not introduce any extra overhead in the GPU memory or in disk. We evaluated Graphie on 7 graphs with up to 1.8 billion edges, and showed that Graphie substantially outperforms X-Stream and GraphReduce.
(e) Friendster-CC

(f) Twitter-CC
Figure 3.7: The number of active partitions vs. the number of transferred partitions.
(a) Friendster

(b) Twitter
Figure 3.8: Performance improvement when using multiple streams.
In this chapter, we will concentrate on another fundamental problem in graph analytics, subgraph isomorphism (also called subgraph matching), which has been applied to many domains. It has been proven that subgraph isomorphism is an NP-complete problem. Thus, subgraph isomorphism usually turns out to be the bottleneck of applications where it presents. Unlike graph traversal problems, the subgraph isomorphism cannot be reformulated into an iterative matrix operation problem. Therefore, subgraph isomorphism is more complicated than subgraph traversals. There have been many efforts devoted to this problem in the past two decades. However, GPU-based subgraph isomorphism systems are relatively rare compared with CPU-based frameworks. The key issue for GPU-based subgraph isomorphism systems is that there will be a huge amount of intermediate instances generated during the matching process. The limited main memory restricts GPU-based subgraph isomorphism systems to small query and data graphs. In addition, most of existing GPU-based frameworks adopt the filtering-joining model and suffer from poor scalability and duplicate computation. The two aspects could severely degrade their performance. Previous work has shown that backtracking is an efficient approach for subgraph isomorphism on CPUs. Nonetheless, the backtracking is quite challenging to implement on GPUs since a recursive kernel call is not supported on GPUs. Moreover, most current GPU subgraph isomorphism frameworks suffer from redundant computation, which could significantly hurt the performance of their systems. To overcome these issues, we designed a new GPU-based subgraph isomorphism system named DGSM, which interprets a query pattern as a series of set operations. This representation, together with other techniques, enables us to remove redundant computation, simulate recursive calls on GPUs, and solve the limited memory issue. In addition, our system efficiently employs the shared memory to improve data parallelism and memory latency. We propose a data structure, VertexSet, to optimize memory bandwidth for set operations in the
matching process. Furthermore, the load imbalance is handled in three ways. We evaluated DGSM by running both labeled and unlabeled queries on seven data graphs and compared the performance of DGSM with that of two state-of-the-art systems, DAF (GPU-based) and GSI (GPU-based). Our experimental results show that our system achieves two orders of magnitude faster than DAF and GSI when answering queries on the data graphs.

4.1 Introduction

Subgraph isomorphism is an essential problem in graph analytics and has a wide spectrum of applications including chemical engineering [54], cybersecurity [51], bioinformatics [43, 8], and artificial intelligence [9]. Notably, subgraph isomorphism is an NP-complete problem. This fact indicates that subgraph isomorphism is generally the bottleneck of any application where it presents. Therefore, an efficient subgraph isomorphism algorithm is demanding and attracts attention from many computer scientists.

Although the first subgraph matching algorithm [68] is first put forward by Ullmann in 1970s using the backtracking approach, computer scientists are trying to design an efficient algorithm for such a problem. Since the last decade, many efforts [22, 7, 21] have been devoted to an efficient algorithm such that it will reduce the searching space and pruning branches as early as possible. Most of them focus on discovering an efficient matching order to achieve these two goals. These systems include TurboISO, CFL, and DAF.

However, these works do not scale well with data graphs due to the data race of updating the auxiliary structures used in these works. Furthermore, these systems need to materialize all the partial results during the matching process and cannot handle some large real-world graphs because of the large amount of intermediate results. It is the reason why DAF [21] and TurboISO [22] are not able to enumerate all the matches and cannot handle large query or data graphs.

Like many other graph algorithms, there are numerous data parallelisms existing in subgraph isomorphism. To employ massive data parallelism in subgraph matching, a few works have implemented GPU-based algorithms to accelerate the matching process. To the
best of our knowledge, almost all of the GPU-based subgraph matching systems adopt the bulk synchronous parallel (BSP) model, which consists of two stages, filtering and joining (or verification). The start-of-the-art systems include Gunrock, GpSM, GSM, and GSI. In these systems, a device needs to shake hands with the host and then allocate memory on the device memory for each iteration since it is not possible to know how many subinstances will be generated by each execution unit beforehand [71, 67, 78]. Gunrock and GpSM do the join computation twice to eliminate write conflicts in the joining phase. However, GSI optimally pre-allocates memory buffers to avoid such duplicate computation. All of these GPU-based systems extend subpatterns by one vertex at a time until the target size is reached. In other words, these systems grow the potential subpattern instances in the fashion of BFS exploration strategy.

Figure 4.1 shows the matching process in the fashion of the BFS exploration when both the query and data graph are the identical triangle. Each red dotted line indicates grid level synchronization between two consecutive iterations. Many other grid level synchronizations inside each iteration have not been shown explicitly in the figure. In the meantime, expensive dynamic device memory allocation is required between each synchronization right after the filtering phase and the joining phase. Besides these synchronizations, it is obvious that the same data graph appears six times in the final results because any permutation of three vertices \( (u_1, u_2, u_3) \) will provide us the same matching subgraph. These duplicates result in both the memory and performance overhead of the application. For a large data graph and a query graph with a plethora of symmetries, one can imagine that a large amount of memory and computation resources are wasted on these duplicates.
According to the above discussion, we summarize the disadvantages of BFS exploration as follows. Firstly, all the subinstances in the current iteration need to be discovered before we can move on to the next iteration. The number of subinstances at one iteration grows exponentially with the size of query and data graphs. Nonetheless, the global memory on a modern GPU device is a few tens GB. This fact results in that the above systems cannot handle subgraph matching with a larger pattern size on large real-world graphs. Secondly, multiple kernels are necessary for their implementations because of the multiple phase BSP computation model, the filtering-joining model. A few grid-level synchronizations are required in one iteration. For a query graph with $N$ vertices, there will be at least $2 \times N$ grid synchronizations. The excessive grid synchronizations can significantly degrade the performance of the matching process for a large query graphs. These downsides are not
avoidable in the BFS exploration strategy.

Including the common disadvantages in the BFS exploration, there is another common issue with their systems. The problem is that the same embedding will be materialized multiple times when there are symmetries present in the target pattern. These duplicates not only waste memory but also lead to unnecessary computation. One possible way to remove these duplicates is integrating an automorphism test in the application. However, the automorphism test is expensive and often leads to a significant performance drop. Therefore, the execution of an automorphism test in each iteration will incur a lot of overhead. It is the reason why GSI retains these duplicates throughout the entire computation.

To overcome the above issues in GPU-based subgraph matching systems, we create a new framework that targets these issues. The contributions of our work are the following:

1. We implemented a new subgraph matching system that is able to do subgraph matching on large real-world graphs for larger query patterns which cannot be done by other GPU-based systems. We adopted the DFS exploration strategy in contrast with other systems which use the BFS exploration strategy. The benefit is that our implementation does not need to materialize all the intermediate results required in the BFS exploration. These intermediate results usually exceed the size of a GPU’s global memory. The DFS exploration renders us the capability to answer queries on large graphs. As mentioned above, the symmetries in a pattern will incur more memory and performance overheads. We utilize a pattern-aware technique to remove duplicates without expensive automorphism tests.

2. We designed a new data structure named VertexSet which can improve memory bandwidth in the matching process. The VertexSet data structure is lightweight and realizes the coalesced memory accesses pattern as much as possible. Therefore, the coalesced memory access pattern is achieved in our system with a minimum cost—compared with other systems.

3. We arranged the edges in a special way so that the filtering phase can be discarded.
This arrangement was done offline only once. Unlike GSI, our system accepts the original edges to obtain the candidate sets with $\mathcal{O}(1)$ complexity (GSI uses auxiliary data structures, Signature and PCSR to aid the filtering phase). The extra storage for the auxiliary data structures will be eliminated in our system. Importantly, we only need to do such an arrangement once and it can be amortized by future computations.

Instead of multiple kernel launches, our implementation only has one kernel launch. It is equivalent to fuse the filtering ($O(1)$ complexity) and joining phases in the BFS exploration into one kernel. Therefore, we completely removed handshakes between CPU and GPU. The global synchronizations existing in other systems were eliminated. There is no dynamic memory allocation at the end of each iteration. We also employed the following techniques to further boost the performance.

1. We used the shared memory in the GPU memory hierarchy to improve the hardware utilization. Without such optimization, the register pressure was remarkable. The overuse of registers will lead to resource underutilization.

2. The load balance issue is mitigated from three perspectives. We first reordered vertex IDs of a data graph based upon the degree. In order to take care of the skewness of the degree, we applied edge-centric mapping in the very beginning. Lastly, we adopted dynamic scheduling to further cope with the load balance.

The rest of the chapter is organized as follows. We first give the definition of subgraph isomorphism and important concepts in this field. The related works are discussed in section 4.3. Section 4.5 presents the challenges of implementing a GPU-based subgraph matching application and our solutions to these challenges. Section 5 proposes our GPU-based subgraph matching system and gives a detailed discussion about the system. The evaluations of our system are shown in the section 4.7 and the conclusion is reached in the section 4.8.
4.2 Background

In this section, we will first present conventions and concepts used in the chapter. Without loss of generality, we assume that only the vertices in pattern and data graphs bear labels. It is straightforward to extend the current work to edge-labeled graphs. We close this section with the definition of two types of subgraph.

4.2.1 Subgraph Matching

Before we give the definition of subgraph isomorphism, we are going to first introduce the notations we use in the thesis. It is worth emphasizing that the terms, subgraph isomorphism and subgraph matching, are interchangeable. A vertex-labeled undirected graph is denoted by $G = (V, E, l, \Sigma)$ where $V$ is the set of vertices, $E \subseteq V \times V$ is the set of edges, $l$ is a surjective mapping from $V$ to $\Sigma$, and $\Sigma$ is the set of labels which is finite and countable. We use $N_G(V)$ and $N_G(E)$ to stand for the number of vertices and edges in a graph $G$. The set of neighbors of a vertex $v$ in graph $G$ is represented by $N_G(v) = \{v' \in V(G) \mid (v, v') \in E(G)\}$. The degree or the size of the neighbor set of $v$ is denoted by $d_G(v)$. Armed with these concepts, we are ready to provide the definition of subgraph isomorphism.

Definition: Given a pattern graph $Q = (V(Q), E(Q), l', \Sigma)$ and a data graph $G = (V(G), E(G), l, \Sigma)$, $Q$ is isomorphic to $G$ if and only if there exists an injective mapping $f$ from $V(Q)$ to $V(G)$, that is $f : V(Q) \rightarrow V(G)$, such that

1. $\forall u \in V(Q), l(u) = l'(f(u))$
2. $\forall (u, u') \in E(Q), (f(u), f(u')) \in E(G)$

Figure 4.2 shows an example of matching a triangle in an input graph. There are 1000 instances of the query triangle in the data graph with the mapping

$\{u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow v_i, \text{ where } i \in \{3 \cdots 1002\}\}$ in this example. One matching instance is also called an embedding.

4.2.2 Two types of embedding

An embedding of the pattern $Q$ in the data graph $G$ can be induced either by a matching vertex subset $V_s(G)$ of $V(G)$ or a corresponding matching edge subset $E_s(G)$. 
We call these two kinds of embedding, vertex-induced matching and edge-induced matching. The vertex-induced embedding is represented by

\[
\{V_v = \{ f(u) \in V(G) \}, \\
E_v = \{ \forall (f(u), f(u')) \in E(G), l, \Sigma \},
\]

(4.1)

while the edge-induced embedding is defined as

\[
\{V_e = \{ f(u) \in V(G) \}, \\
E_e = \{ (f(u), f(u')) \in E(G), \forall (u, u') \in E(P), l, \Sigma \}. 
\]

(4.2)

The vertex-induced embedding includes all the edges connecting the mapped vertices. However, the edge-induced embedding cares about whether every single edge in the query graph has a counterpart among mapped vertices. The difference between two kinds of subgraphs can be seen in the case of searching wedges in the triangle data graph shown in Figure 4.3.
For convenience, we assume there is only one element in $\Sigma$. This example has six potential mappings as follows:

\begin{align*}
  & u_1 \mapsto v_1 \quad u_2 \mapsto v_2 \quad u_3 \mapsto v_3, \quad (4.3) \\
  & u_1 \mapsto v_1 \quad u_2 \mapsto v_3 \quad u_3 \mapsto v_2, \quad (4.4) \\
  & u_1 \mapsto v_2 \quad u_2 \mapsto v_1 \quad u_3 \mapsto v_3, \quad (4.5) \\
  & u_1 \mapsto v_2 \quad u_2 \mapsto v_3 \quad u_3 \mapsto v_1, \quad (4.6) \\
  & u_1 \mapsto v_3 \quad u_2 \mapsto v_1 \quad u_3 \mapsto v_2, \quad (4.7) \\
  & u_1 \mapsto v_3 \quad u_2 \mapsto v_2 \quad u_3 \mapsto v_1. \quad (4.8)
\end{align*}

According to the definition 4.1, there is no vertex-induced embedding in these six mappings since all of the six mappings will lead to the same subgraph,

\begin{align*}
  V &= \{v_1, v_2, v_3\} \\
  E &= \{(u_1, u_2), (u_1, u_3), (u_2, u_3)\}. \quad (4.9)
\end{align*}

The query pattern, wedge, requires that an embedding induced by a mapping should not contain the missing edge in the query graph. Therefore, there is zero wedge in the triangle data graph when vertex-induced embedding is concerned. However, these six mappings will result in six edge-induced embeddings since each edge in the query graph has a correspondence in the data graph for each mapping.

As there is massive parallelism in graph traversals, so too are there plenty of data parallelisms existing in subgraph isomorphism. It makes GPU a suitable platform to
improve the performance of subgraph matching since thousands of threads can be launched simultaneously on a GPU device. Designing an efficient GPU-based subgraph matching application is more challenging than that for graph traversals due to several facts. Utilizing GPU to accelerate subgraph isomporhism could be a double sided sword. Therefore, a programmer should be familiar with GPU architecture features in order to avoid pitfalls in GPU-based subgraph isomorphism. Please refer to Chapter 2 for details of modern GPU architectures.

4.3 Related Work

The original Ullmann algorithm [68, 36] needs to try all the possible mappings for a vertex \( u \) in a query graph and figure out a complete mapping of the query at the end. Therefore, it turns out to be the slowest subgraph matching algorithm so far. VF2 [12] improves the performance by imposing more pruning constraints than the Ullmann algorithm. VF2 requires that the next query vertex to be matched has to be connected to the query vertices which are already matched.

A significant progress in subgraph matching has been witnessed since the beginning of this century, especially the past decade. Several proposals were made to speed up the subgraph matching problem on CPUs. BoostISO [56] exploits vertex relationships in the data graph to reduce duplicate computation and speed up subgraph isomorphism. Nevertheless, most works strive to search for a better matching order limiting the search space. They can be classified into two categories, vertex order and path order. VF2 and QuickSI [62] are two typical systems in the former category. QuickSI selects the most infrequent query vertex as the next query vertex to be matched. The well-recognized systems falling into the second category include Turbo\textsubscript{ISO}, CFL, and DAF. Turbo\textsubscript{ISO} [22] builds a spanning tree from a query graph by running a BFS on the query graph and establishes a matching order based upon the size of potential matches of a path. The edges in a query graph are classified into two classes depending on the condition whether an edge in a query graph is present in the corresponding spanning tree or not. The edges which are
not present in the query tree are called non-tree edges. In TurboISO, the non-tree edges will be taken into account at the end. However, non-tree edges usually have more pruning power than normal edges. Therefore, such a construction leads to poor pruning power in TurboISO. To cope with the symmetries in the pattern graph, TurboISO rewrites the query graph to a neighborhood-equivalence-class tree.

To overcome the issue with non-tree edges, CFL [7] decomposes a query graph into three parts which are core, forest, and leaf. The core is the subgraph with the minimum vertices containing all the non-tree edges. CFL will matching the core part first to apply the pruning power of non-tree edges as early as possible. After the decomposition, CFL designs an auxiliary data structure, known as compact path-index, to facilitate the matching.

DAF [21] takes a different route from the above two systems. It first builds a DAG from a query graph using BFS. The non-tree edges, whose direction depends on the order of vertices, will appear in the DAG. Then DAF will construct an auxiliary data structure, CS, to enumerate the embeddings. DAF proposes an adaptive matching order based on the path size, and it introduces failing sets to reduce the searching space. Both DAF and TurboISO prefer the path order [14] over the vertex order.

Although these three algorithms prove their efficiencies to some extent, these algorithms are not easy to scale up to real-world graphs, have poor data parallelism, and suffer severe load balance issues.

To employ data parallelism in subgraph matching, a few research groups have worked on GPU-based implementations. Most of these GPU systems adopt the two-phase BSP models. The state-of-the-art GPU systems include GSM [71], GunrockSM [72], GpSM [67, 66], and GSI [78]. GSM is composed of filtering and verification phases. First, the filtering process prunes out those candidate vertices which will not contribute to the final solution through some constraints. Then the verification phase follows. The traditional verification is a backtracking-based algorithm which is challenging to implement on GPUs. Therefore, GSM adopts all source BFS traversals from every node in the data
GSM utilizes two optimizations to limit the memory consumption of intermediate results. First, GSM does a compaction on the updated candidate set before writing to partial results. Second, GSM encodes a potential subinstance in each iteration by hashing node ID tuples to a value to reduce memory complexity. In addition, GSM uses a k-look-ahead to optimize the verification process.

GunrockSM, GpSM, and GSI consist of filtering and joining phases. These systems extends the subpatterns by adding one vertex at a time through the BFS exploration. First, they compute the next query vertex to be match based on the topology information of the query graph and obtain the candidate set through the filtering process. Then they calculate valid candidates and join them to get updated subembeddings. Such an approach is easier to utilize data parallelism and to take care of load balance in each phase. GSM, Gunrock and GpSM adopt a two-step output scheme to enable writing partial results to GPU memory in a massively parallel way. To make it clear, we demonstrate the two-step output scheme with the simplest query pattern in Figure 4.4. Let’s assume that the query graph is a wedge with three different labels in Figure 4.4a and the data graph is shown in Figure 4.4b.

It can be seen in Figure 4.4c that the two-step output scheme performs the join twice to enable parallel writes. GSI avoids duplicate computation by a Prealloc-Combine approach based on joining vertices.

Besides the above matching strategies, there are some frameworks using the join approaches: query-edge-at-a-time [26, 77] and query-vertex-at-a-time [3, 44, 48, 4, 49]. These frameworks are popular in DBMS and suitable for a distributed environment because they need to store instances of many subpatterns. It has been shown that the first approach is suboptimal and its runtime is asymptotically worse than that of the second one [5].
Matching order: $u_1 \rightarrow u_2 \rightarrow u_3$

(a) Query Graph

(b) Data Graph

(c) Pattern extension where two join operations are separated by dashed line

Figure 4.4: Two-step output scheme in Gunrock and GpSM
4.4 Motivations

We have seen several successful GPU-based subgraph matching frameworks in the Section 4.3. However, all of these GPU frameworks have drawbacks which hurt their performance substantially. The drawbacks in these systems motivate us to revisit the implementation of GPU-based subgraph matching.

First of all, the biggest disadvantage of this model is a large amount of intermediate results needs to be materialized in the global memory during matching. Due to the limitation of the global memory, it makes impossible for these systems to handle large query patterns and data graphs. These systems do not scale well on graphs, especially when the unlabeled graph is concerned. For instance, GSI cannot even deal with a pattern with four vertices on a relatively small real-world graph like wiki-Vote. Even for labeled graphs, patterns in motif-4 cannot be queried on Livejournal using GSI. All of the GPU-based systems mentioned above generate a large amount of intermediate results no matter what model they adopt, even though GSM uses a value to encode the subembeddings. Therefore, these frameworks have poor scalability.

Secondly, all of these GPU-based systems suffer from redundant computation. The redundant computation originates from three aspects including duplicate join computation in Figure 4.4, symmetries, and connectivity verification. Gunrock and GpSM need to do the join computation twice in order to eliminate memory conflicts in parallel write. The purpose of the first join computation is to calculate how many instances one warp will generate. A prefix scan follows immediately to calculate the starting address for each warp. Then the second join will update the match table and allow parallel writes to the memory. GSI avoids the duplicate computation by conservatively preallocating memory to enable parallel writes. GSI first calculates the maximum possible matches that each warp might generate using candidates’ information, completes an exclusive prefix-sum scan on these potential matches, and then preallocates a memory buffer for parallel writes of the updated matching table.
In addition, the four systems do not remove the identical results coming from the symmetries in a pattern graph. To remove redundant results, an automorphism test is needed at the end of each extension. The four systems do not incorporate an automorphism test in their implementation because such a test is expensive, and there is no efficient algorithm for the test. They will carry these duplicates throughout the whole matching. These duplicates will definitely waste memory and computation resources. For clarity, Figure 4.5 is an example where swapping $u_3$ and $u_4$ leaves the pattern invariant. The outcome of this symmetry is that the same subgraph induced by vertices $v_1, v_2, v_3$ and $v_4$ appears two times in the results by exchanging the mapping of $u_3$ and $u_4$.

The third kind of redundant computation can be demonstrated by the same query in Figure 4.5, where both $u_3$ and $u_4$ have the common neighbors $u_1$ and $u_2$. According to Algorithm 3 in GSI, the same connectivity check for $u_3$ and $u_4$ will be done twice.

Thirdly, GSI also introduces two auxiliary data structures called Signature and PCSR to facilitate the filtering phase. The goal of Signature and PCSR is to make the filtering step easier for GPUs. However, there are costs associated with these data structures. For example, both data structures need extra memory overhead and waste memory bandwidth to look up the candidates. These two structures require a preprocess on the input data graph. This preprocess might take a while to complete on the large real-world data graphs,
like Patents and Livejournal.

Fourthly, many grid-level synchronizations are necessary for large queries in the existing GPU-based systems because they make use of the BFS exploration approach. These systems demand at least three kernels to fulfill filtering, joining, and writing in parallel. There exist strong data dependencies among these kernels. Therefore, the three operations are severely serialized. In addition, it incurs a lot of overhead to do memory allocation, control transfer, and grid synchronizations among consecutive kernels for large query graphs.

Lastly, these systems depend on inefficient data structures to store matching instances throughout the whole matching process. The most popular data structures in subgraph matching and graph mining (a similar but easier task to subgraph matching) are the matching table [70, 27, 78] or a tree-like data structure [11, 22, 21]. Two data structures can be applied in GPU-based frameworks. However, neither of them are GPU friendly. Each entry in a matching table corresponds to an embedding or matching instance. GSI stores matching instances in a matching table. Although a matching table guarantees coalesced memory transactions, it could waste a lot of memory space to store the vertices which are shared by multiple instances. This fact is reflected in Figure 4.5, where all the embeddings share two vertices: $v_1$ and $v_2$. Both $v_1$ and $v_2$ will appear $2 \times \binom{1000}{2}$ times in the matching table because of symmetries. Such a storage layout wastes memory for saving the common vertices shared by plenty of instances. Even though GSI removes duplicate memory loads from instances sharing common vertices inside a thread block, the redundant memory accesses among thread blocks remain unsolved, especially for some hot vertices. Another disadvantage is that a matching table needs to write previous matching vertices to the updated table. Thus, the matching table wastes memory bandwidth.

Pangolin proposes a tree-like data structure, called an embedding list. The embedding list needs additional information on each node in the structure to trace back previous vertices in the same embedding. Even though the embedding list eliminates the memory bandwidth problem in the matching table, it makes the memory accesses of an instance
uncoalesced. TurboIS and DAF utilize tree structures, representing an embedding. The benefit is that the redundant storage space for common vertices is optimized. However, the tree structures proposed in their work suffer from load imbalancing, low-data parallelism, and high-memory latency.

These issues in the existing GPU-based frameworks motivate us to revisit subgraph matching on GPUs.

4.5 Challenges in GPU-based subgraph isomorphism

4.5.1 Challenge I

A typical size of the GPU main memory is about a few tens GB. However, the intermediate results of some large graphs can reach several TBs. GPU systems based on the filtering-joining strategy have to discover all the subembeddings in the current iteration in order to proceed to the next iteration. This strategy is known as the BFS exploration. The number of subembeddings is exponentially proportional to the number of vertices in data graphs, and it turns out to be the biggest challenge in a GPU-based subgraph matching framework.

4.5.2 Challenge II

A massive amount of data should be fed to threads so that higher hardware utilization and data parallelism are guaranteed. Therefore, we ask, how do we make memory accesses as efficient as possible; this is critical to the performance of a system. The best option is to realize coalesced memory accesses because this pattern can give rise to optimal memory bandwidth. However, the irregularity of a graph’s data often renders coalesced memory accesses difficult.

4.5.3 Challenge III

In most GPU-based subgraph matching implementations, multiple threads work collectively on a subembedding. In order to avoid warp divergence, we assign a warp to work on an instance— that is, a warp will share some candidate sets in the matching process. A candidate set should contain at least the following information: the size of the
set, the source vertex ID, and the starting address of a set. So the minimum memory overhead of a candidate set is no fewer than 16 Bytes. Putting the shared candidate sets in register obtains a higher bandwidth and lower latency. Nevertheless, this scheme usually causes register spilling for large queries. Overuse of the register will lead to poor thread occupancy, data parallelism, and hardware utilization. If we simply cache the metadata of these sets in the register, we will not achieve 100% thread occupancy even for a pattern with 4 vertices on a graph like Mico. Storing the metadata in the global memory is another option. In fact, the memory latency generally outweighs the thread occupancy in this scenario. Therefore, how do we maintain higher thread occupancy and lower memory latency for large pattern and data graphs? This question is important to the performance of a system.

4.5.4 Challenge IV

It has been shown in TurboISO [22] that backtracking is a successful approach for subgraph matching. Algorithms 7 shows the pseudo-code of a backtracking implementation similar to VF2. However, a GPU-based backtracking algorithm is quite challenging since a recursive kernel function is not supported on GPU devices. It would be impossible to directly transform such algorithm into a CUDA kernel.

An alternative and easy solution to backtracking is the BFS exploration strategy adopted by the other GPU-based systems. The BFS exploration’s advantage is that the load imbalance is relieved by one separate kernel with different launch configurations at each iteration. Nonetheless, the BFS exploration suffers from a lot issues mentioned in Section 4.3, which will considerably kill the performance of a system.

4.5.5 Challenge V

The irregularity of a graph not only incurs uncoalesced memory accesses but also causes load balance issues. The load balance is always a main concern in a parallel application’s performance. The load balance is more difficult in GPU-based frameworks than CPU-based frameworks because the thread synchronization and memory sharing on
GPUs are more complicated than CPUs.

**Algorithm 7: Subgraph Isomorphism**

**Input:** $G(P), G(g)$

**Output:** $\text{Results}$

1. **Function** $\text{Main}(G(P), G(D))$:
   1. Initialization: $\text{Results}, M(p)$;
   2. $\text{subMor}(M(p), G(P), G(D), \text{Results})$;
   3. return $\text{Results}$;

2. **Function** $\text{subMor}(M(p), G(P), G(g), \text{Results})$:
   1. if $M(p)$ and $G(P)$ have the same number of vertices then
      1. Append($M(p), \text{Results}$);
   2. else
      1. $CS = \text{getNextCandidate}(M(p), G(P))$;
      2. /* The candidate set for next vertex to be matched. */
      3. for $u$ in $CS$ do
         1. $M(p') = \text{Extend}(M(p), u)$;
         2. $\text{subMor}(M(p'), G(P), G(g), \text{Results})$;
         3. $\text{backTracking}(M(p'))$;
   4. end
   5. return

4.6 DGSM

In this section, we will present our GPU-based subgraph isomorphism implementation, DGSM, in detail. Also we will explain how we sort out the challenges in section 4.5.

4.6.1 Solution to Challenge I

As pointed out in Section 4.5, the source of large intermediate results is the BFS exploration strategy. Instead of the BFS exploration, we choose the recursive DFS
exploration strategy to solve the large memory requirement issue. The benefits of the DFS exploration strategy are threefold. First, the number of potential instances are limited in the DFS exploration. Not all the entire subembeddings in the data graph are necessary to move on to the next iteration. Only the subembeddings already discovered by the warps need to be materialized in the memory. Second, many grid-level synchronizations are eliminated. A warp will continue to extends its own subembeddings until the last query vertex is finished. Therefore, no communications among warps or thread blocks are required in the DFS exploration. The majority of grid-level synchronizations in GSI are not present in our implementation. Third, we fuse the kernels in the filtering and joining phases into a single one. So no control transfer is needed between the host and the device in order to make dynamic memory allocations inside each iteration. With the help of the DFS exploration, we are able to preallocate a memory buffer before the matching starts on GPUs.

4.6.2 Solution to Challenge II

We propose a data structure, called VertexSet, shown in Algorithm 8 to store the information of a candidate set in order to achieve coalesced memory accesses for set operations. All the possible mappings of a query vertex \( u \) are stored in a continuous chunk of memory. Different warps will maintain their own VertexSets. There will be no communication among warps. This structure also makes the backtracking process and references to the VertexSets in previous iterations a lot easier. Most importantly, it supports us in reusing some of the VertexSets to avoid duplicate connectivity checks in GSI. Therefore, there would be no extra memory overhead, and no memory bandwidth will be wasted with the help of the VertexSet structure.

**Algorithm 8: VertexSet**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Struct <strong>VertexSet</strong> contains</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
4.6.3 Solution to Challenge III

To minimize warp divergence, the threads in a warp work collectively on the embeddings emanated from a candidate vertex. Therefore, they are going to share and work on the same VertexSets. How will sharing these VertexSets affect the performance? Caching the metadata of these VertexSets into the global memory or registers is not the best solution. If they are stored in thread private registers, the lowest latency is achieved, but the thread occupancy might be reduced significantly for large patterns because each thread needs 16 bytes for one VertexSet. Algorithm 9 shows that 6 VertexSets are required for a pattern with 4 vertices. The computation resource will not be fully saturated without register spilling. There will be many more VertexSets for a larger pattern, and this large number of VertexSets will lead to either register spilling or lower thread occupancy. Putting these VertexSets in the global memory will not reduce thread occupancy, while it incurs higher memory latency to access them. Both options will not help to improve the performance for large query patterns. However, GPUs expose a programmable L1 cache, called shared memory, to us. The shared memory can be used to share data among threads in a thread block, so we can resort to the shared memory to reduce the register pressure mentioned above. Caching VertexSets in the shared memory will help us to get better occupancy without sacrificing too much latency.

4.6.4 Solution to Challenge IV

The backtracking, or the DFS approach, cannot be ported to GPUs since a recursive kernel is not supported on GPUs. However, we can simulate the backtracking process with the help of the data structure VertexSet and the matching order of a query pattern. DGSM matches one query vertex at a time, so the matching process can be abstracted as a nested for loop. The matching process of the pattern 4.5a can be written as Algorithm 9, where \( N(m_0, C) \) stands for the neighbors of \( m_0 \) with label \( C \), \( \cap \) is the intersection operation of two sets, and \(-\) denotes the difference of two sets. As seen in Algorithm 9, we enforce the partial order through a bound on the target VertexSet \( c_6 \). The VertexSet set \( c_4 \)
will be used for matching both $u_3$ and $u_4$. Thus, it should be computed in the nested for loop as early as possible to avoid redundant computation. In this case, it is computed inside the second for loop. Therefore, we did not waste memory and hardware for duplicate subembeddings and compute the set intersection twice, like GSI, for the pattern in Figure 4.5a. The VertexSet $c_4$, which is the intersection of the neighbor list of $m_0$ and $m_1$ with label $B$, has been calculated in the second for loop. In the third for loop, we will use the result by referencing VertexSet $c_4$. It should be emphasized that the overhead of the analysis on a query graph is negligible since the query graph is usually small, and the analysis is done on the host side.

**Algorithm 9: Backtracking Unrolling**

```
1   VertexSet $c_0 = \text{getCandidate}(A)$;
2   for auto $m_0 \in c_0$ do
3       VertexSet $c_1 = N(m_0, C)$;
4       VertexSet $c_2 = N(m_0, B)$;
5       for $m_1 \in c_1$ do
6           VertexSet $c_3 = N(m_1, B)$;
7           VertexSet $c_4 = c_2 \cap c_3$;
8           for $m_3 \in c_4$ do
9               VertexSet $c_5 = N(m_3, B)$;
10              VertexSet $c_6 = \{ m \in c_4 - c_5 \text{ and } m < m_3 \}$;
11         end
12     end
13 end
```

Although the recursive call is not supported on GPUs, we can implement the DFS approach if we view the backtracking process as a nested for loop and generate a plan based on the nested for loop. A nest for loop can be simulated with an array of counters on GPUs. The plan maintains the topology of subpatterns at each step, specifies the matching order and partial orders among vertices of a query graph, and has a one-to-one correspondence to a nested for loop. Once such a plan is generated, it will be moved to a GPU to guide the computation on the GPU.
4.6.5 Solution to Challenge V

The load balance in the DFS exploration can be very complicated. A good understanding of the cause of load imbalance helps us to find better solutions to this issue. DGSM targets the origin of load imbalance in subgraph matching that is the skewness of vertex degrees. With the knowledge of source of load imbalance, we cope with the issue in subgraph matching with three techniques. The first solution is that we reorder the vertex ids according to the degrees. The more neighbors a vertex has, the less its id will be. A graph is stored in the most compacted CSR format. We further sort the neighbors of a vertex based on its label. The neighbors of a vertex with the same label are saved continuously in the edge list and are sorted in ascending order in that memory chunk. Figure 4.6 shows the details of how we organize an edge list. This organization of edges will not only improve the load balance issue, but it will also validate the design of the VertexSet structure where GPU memory access pattern is concerned. It is the reason why we do not need to preallocate memory buffers for VertexSets, which are neighbors of vertices, and we are able to assign them to addresses of the edge list on the fly. Notably vertex reordering can be done offline, and it is done only once.

<table>
<thead>
<tr>
<th>$N(v_1)$</th>
<th>$N(v_2002)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_3 \cdots v_{2002} v_2 \cdots$</td>
<td>$v_1 v_2$</td>
</tr>
</tbody>
</table>

$N(v_1, B)$ $N(v_1, C)$

$N(v_{2002}, A)$ $N(v_{2002}, C)$

Figure 4.6: Edge List

Secondly, a parallel strategy is data parallelism on the the set $c_0$ in Algorithm 11. This strategy might suffer severe load imbalance due to the skewness of vertex degrees shown in the Figure 4.7. The red wavy arrows indicate the data parallelism on the vertices in the VertexSet $c_0$. The workload assigned to the thread $T_1$ is 100 times more than that of $T_2$. 
However, perfect load balance is achieved if each edge with labels $A$ and $C$ is assigned to a thread indicated by green wavy arrows. In such a case, each thread will have the same amount of work.

Thirdly, we adopt dynamic scheduling to deal with the potential load balance in deeper iterations. Because the edge list is sorted according to the degrees of Vertices, the static scheduling possibly always assigns a heavier workload to a certain warp in each round. Thus, the static scheduling could lead to a severe load imbalance and performance drop. This load imbalance issue can be attenuated by the dynamic scheduling.

Putting these solutions together, we present DGSM in Algorithm 10.

![Diagram of load imbalance](image-url)

Figure 4.7: Example of load imbalance
Algorithm 10: GPU-based Subgraph Isomorphism

Input: $G(P), G(g)$
Output: Results

1 Function Main($G(P), G(D)$):
   2 Initialization: Results;
   3 Load($G(g)$);
   4 Preprocess($G(g)$);
   5 allocateMemory($G(P)$);
   6 $plan = generatePlan(G(P))$;
   7 copyH2D($plan$);
   8 kernel<<<<<grid, block >>>>$(plan, G(g), Results$);
   9 copyD2H(Results$);
10 return Results;

11 Function kernel($plan, G(P), G(g), Results, buffer$):
   12 $wid =$ getWarpId$();$
   13 $tid = laneid();$
   14 initSharedMem;
   15 init(VertexSet, buffer);
   16 if $tid == 0$ then
      17      $eid =$ atomicAdd$($edge$);$n
   18      while $eid != num_edges$ do
   19         if $eid == num_edges$ then
   20            break;
   21         Extend($plan, eid, VertexSets$);
   22      if $tid == 0$ then
   23         $eid =$ atomicAdd$($edge$);$n
   24      end
   25 16 end
26 return

4.6.6 Other Optimizations

DGSM has two main components, the plan generator on line 6 and the matching kernel on line 8 in Algorithm 10. The main purpose of the plan generator is to devise a matching order which can prune away unpromised branches as soon as possible. The matching order is determined by an analysis on a pattern graph only. The paper [41] explains the details about how the optimal matching order is generated. This approach considers multiple paths through a query vertex at a time during the extension, while DAF proceeds with one path at a time. The benefit of our approach is that our system will backtrack immediately once one path containing the query vertex fails. DAF might waste
computation on some full paths before it encounters a failing path. A non-tree edge is taken care of as soon as two query vertices connected by it are discovered. Moreover, the ordering strategy serves to remove redundant computations. In addition, it should be stressed that redundant computation comes from three perspectives. First of all, there are duplicate instances when symmetries exist in the pattern graph. One way to get rid of these duplicates is to do an automorphism test once all the subembeddings in the current iteration are obtained. However, an automorphism test is very expensive. An efficient parallel automorphism test on both CPUs and GPUs is still an open question for now. In order to avoid the expensive automorphism test, we will do an analysis on the pattern graphs first to find all the equivalent classes. An equivalent class means that all the vertices belonging to it are interchangeable. Then we impose a partial order on the query vertices in each equivalent class to break the symmetries in the pattern. The query graph in Figure 4.5a is an example. As we have already pointed out, there are two vertices, $u_3$ and $u_4$, and these vertices are interchangeable. Exchanging the mapping of these two vertices will lead to the same subgraph. If we enforce the partial order shown at the bottom of Figure 4.8 on the two query vertices in the pattern, we can remove duplicate embeddings of the query in a data graph.

![Figure 4.8: Pattern with Partial Orders](image)

Figure 4.8: Pattern with Partial Orders
In this case, only the mapping \( \{ u_1 \to v_2, u_2 \to v_1, u_3 \to v_4, u_4 \to v_3 \} \) is valid, while the other mapping \( \{ u_1 \to v_2, u_2 \to v_1, u_3 \to v_3, u_4 \to v_4 \} \) violates the partial order requirement \( u_3 < u_4 \). Here, we assume that a subscript in the data graph is the ID for that vertex.

Secondly, the DFS strategy allows us to preallocate a memory buffer for the VertexSets needed by each warp, even though large queries are concerned. Each VertexSet owns a continuous chunk in the memory buffer so that a coalesced memory access is warranted. This preallocation is only done once before the kernel starts. It will free us from the duplicate join computation in Gunrock and GpSM and reduce memory allocation overhead for each query vertex in GSI.

Thirdly, there exists duplicate connectivity computations in some patterns. The query vertices \( u_3 \) and \( u_4 \) in Figure 4.8 are both connected to the vertices \( u_1 \) and \( u_2 \). GSI will do set intersections with matched vertices of \( u_1 \) and \( u_2 \) twice to get the candidate embeddings, one for \( u_3 \) and another for \( u_4 \). With the help of our VertexSet data structure, we only need to compute the set intersection once on line 7 in Algorithm 11. We cache the candidates for \( u_3 \) and \( u_4 \) in the VertexSet \( c_4 \) when \( u_3 \) is matched. We further directly retrieve the candidates for \( u_4 \) from \( c_4 \) later on. Besides redundant computation, our plan already encodes the connectivity among query vertices. An unique subpattern is ensured at each matching step. Therefore, automorphism tests can be safely avoided.

After the GPU receives a plan, the kernel will be launched right away. We allocate a warp to collectively find out some embeddings with the purpose of reducing warp divergence. Importantly, we need to create seven VertexSets for each warp for a pattern graph with four vertices. The size of the register file on our GPU is 256KB, and each SM supports up to 2048 threads. Such a configuration reflects that each thread can use, at most, 32 registers to achieve 100% occupancy. Undoubtedly, the registers will be rapidly consumed up by a larger query graph. The application will have either lower occupancy or the register spilling for large queries. Both cases are not what we expect. The former results in lower data parallelism, while the latter has higher memory latency. To ease register pressure, we use the shared memory to cache the metadata of these VertexSets for
fast accessing. These VertexSets will be visited millions of times, so it is ideal to cache the metadata in the shared memory. We also use the shared memory to store the counter for each VertexSet (c1 and c4 in this case), which we will iterate through. Algorithm 11 illustrates how VertexSets and counters are implemented in our system. The variable buffer in Algorithm 11 is preallocated in the global memory to store the results of each VertexSet. Line 8 sets all the counters to 0. The number of registers is reduced down to two if the shared memory is used to store the metadata of the VertexSets for the query in Figure 4.5a. Since the shared memory has the same latency as L1 cache, the performance is not sacrificed much. In our implementation, we only preallocate memory buffers for necessary VertexSets. In our example, c1, c2, c3, and c5 do not need separate memory buffers since the buffers in these VertexSets can directly point to the neighbor list.

Figure 4.9 shows the layout of six VertexSets for the query in Figure 4.5a per thread block. The solid arrows indicate those VertexSets are preallocated and initialized in the kernel, while dashed arrows mean that the VertexSets will be set on the fly.

Figure 4.9: Arrangement of VertexSets
Bearing the organization of the edges shown in Figure 2.7 in your mind, it would be reasonable to see why we could use the neighbor list as the buffers for some VertexSets.

**Algorithm 11: Shared memory usage snippet**

```
1 extern __shared__ int shmem[];
2 VertexSet* vs_ptr = (VertexSet*) &shmem[warp_vs_offset];
3 uint32* cnter_ptr = (uint32_t*) &shmem[warp_cnter_offset];
4 for tid < numVertexSets do
5     vs_ptr[tid].data = &buffer[warp_vs_idx_offset];
6     vs_ptr[tid].vid = -1;
7     vs_ptr[tid].size = 0;
8     Initialize(cnter_ptr, num_cnter, 0);
9     __synccwarp();
```

Incorporating all these techniques together into Algorithm 10, we implement our subgraph matching framework, which is able to do both vertex-induced and edge-induced matching for labeled and unlabeled queries. Algorithm 9 gives the procedure of vertex-induced matching. For the edge-induced matching, the only modification is the difference operator of two sets.

### 4.7 Evaluation

We present our experimental results in this section. Both vertex-induced and edge-induced queries are evaluated. We will show the performance of both matching schemes using patterns in motifs and cliques. Then, we will demonstrate the performance benefits of each technique we introduced in section 4.6 after integrating them. Finally, we will assess the performance of our system by comparing against two the state-of-the-art subgraph matching systems, DAF (CPU-based) and GSI (GPU-based), on some random query graphs. In this section, each query is executed 10 times, and the average is reported. The standard deviation of each query’s performance is not more than 3% of the average. Therefore, we do not show them for each query.

#### 4.7.1 Experiment Setup

We are going to evaluate our system on seven real-world graphs in Table 4.1. All of our experiments are conducted on a host with Intel(R) Xeon(R) CPU E7-4830v3 2.10GHz.
24-core CPUs and 256GB DRAM equipped with a NVIDIA TITAN V GPU 12GB of RAM. The GPU has an array of 80 streaming processors, and each SM can sustain up to 2048 threads. We use the NVCC compiler version 11.0.221 (g++ version 7.5.0) with O3 to compile all the programs. The operating system is Ubuntu Linux 18.04 with Linux kernel version 5.4.0. The GSI source code is available and can be obtained from its codebase on GitHub. DAF is not open-sourced, but the executables are available on GitHub. The default number of threads is set at 24 when DAF is concerned.

| Dataset ($G$) | $|V(G)|$ | $|E(G)|$ | $d_{max}(G)$ |
|----------------|---------|---------|-------------|
| Enron          | 36692   | 367662  | 1383        |
| Amazon         | 1134890 | 1851744 | 549         |
| DBLP           | 317080  | 2099732 | 343         |
| Mico           | 96638   | 2160312 | 1359        |
| Patents        | 3774768 | 33037894| 793         |
| Livejournal    | 4846609 | 85702474| 20333       |
| wiki-Vote      | 7115    | 201524  | 1065        |

### 4.7.2 Effectiveness of Our Techniques

We will first show the effectiveness of the techniques on unlabeled patterns because an unlabeled query has more symmetries and the number of embeddings than a labeled one for a certain query graph. The baseline of our implementation is that all the VertexSet metadata are stored in registers, data parallelism is carried out on vertices, and the static scheduling is chosen. Without using the shared memory to cache the VertexSet metadata, only 50% thread occupancy can be achieved for a size-4 pattern because of the aggressive usage of registers in the baseline implementation. Table 4.2 presents the performance of the baseline when answering queries in motif-4 on the graphs in Table 4.1. The convention is that PA, ST, RE, CR, TT, and CL stand for the patterns path-4, star-4, rectangle, chordal rectangle, tailed triangle, and clique-4 in motif-4.
Table 4.2: Baseline Performance (Second)

<table>
<thead>
<tr>
<th>Graph</th>
<th>PA</th>
<th>ST</th>
<th>RE</th>
<th>CR</th>
<th>TT</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>34.9</td>
<td>1288.74</td>
<td>8.95</td>
<td>6.9</td>
<td>150.1</td>
<td>4.03</td>
</tr>
<tr>
<td>Patents</td>
<td>2.83</td>
<td>3.99</td>
<td>2.67</td>
<td>2.61</td>
<td>2.62</td>
<td>2.78</td>
</tr>
<tr>
<td>Mico</td>
<td>0.37</td>
<td>6.43</td>
<td>0.215</td>
<td>0.48</td>
<td>10.97</td>
<td>0.23</td>
</tr>
<tr>
<td>DBLP</td>
<td>0.2</td>
<td>0.41</td>
<td>0.2</td>
<td>0.19</td>
<td>0.83</td>
<td>0.21</td>
</tr>
<tr>
<td>Amazon</td>
<td>0.21</td>
<td>0.63</td>
<td>0.2</td>
<td>0.2</td>
<td>0.28</td>
<td>0.22</td>
</tr>
<tr>
<td>Enron</td>
<td>0.47</td>
<td>3.98</td>
<td>0.13</td>
<td>0.16</td>
<td>0.74</td>
<td>0.07</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>0.12</td>
<td>2.81</td>
<td>0.05</td>
<td>0.04</td>
<td>0.43</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Figure 4.10: Speedup utilizing shared memory

To reveal the benefit of each technique, we present the speedup of implementations, incorporating different techniques to the baseline. Figure 4.10 shows the speedup after implementing the shared memory optimization. There is a performance gain for each query pattern. Some queries can obtain outstanding speedups on Patents, DBLP, and Amazon because of the higher thread occupancy. The speedup on patterns Q1 and Q2 are not as great as other patterns. The reason for this is that computations on hot vertices dominate the execution and lead to load imbalance. Figure 4.10 shows that the shared memory
optimization is an order of magnitude faster than the baseline on average. Figure 4.11 displays the speedups of the dynamic scheduling implementation. It is apparent that the speedups have been improved with the dynamic scheduling when looking at Figure 4.10 and Figure 4.11. The skyline of the speedups in Figure 4.11 is the similar to that in Figure 4.10. This fact tells us that although the dynamic scheduling is helpful for the load imbalance to a certain extent, it alone cannot solve the load imbalance from hot vertices.

![Figure 4.11: Speedup of Dynamic Scheduling](image)

Fig. 4.11 illustrates the final speedup after we adopt the following techniques, including shared memory, dynamic scheduling and the edge mapping shown in Fig. 4.7. Table 4.4 lists the performance of vertex-induced matching, incorporating all the aforementioned techniques for the same patterns in Table 4.2. As mentioned earlier, our system cannot only perform vertex-induced matching, but it also accomplishes edge-induced queries. The edge-induced query results for the same patterns are displayed in Table 4.3.
The difference between vertex-induced and edge-induced matching is whether the missing edges between vertices are verified or not. The vertex-induced matching has to validate missing edges, while the edge-induced matching does not need this process.

Table 4.3 introduces the performance of the edge-induced matching when the vertices in a query are not more than four. In general, the edge-induced matching outperforms the vertex-induced matching. This fact can be witnessed by comparing Table 4.4 to Table 4.3. However, the edge-induced matching will discover more patterns than the vertex-induced one. The performance discrepancy between two matching approaches depends on the
balancing between the missing edge check and the difference in the number of discovered instances. There are only a few edge-induced queries slower than the corresponding vertex-induced query. The reason is that those queries have a larger ratio of the number of edge-induced instances to the vertex-induced instances. Since the triangle query is trivial, we combine the performance of the wedge and triangle together in motif-3.

Figure 4.13 illustrates the speedups of the edge-induced matching to the baseline, and Figure 4.14 shows the speedups of the dynamic scheduling to the static scheduling for edge-induced matching. For clique-4, no missing edge exists. There is no difference between the edge-induced and vertex-induced matching. Therefore, we do not show the speedups for the clique-4 pattern. We truncate the speedup of Q4 on Enron down to 2 in Figure 4.14, and its actual speedup is 14.
Table 4.4: Vertex-induced Matching Performance (ms)

<table>
<thead>
<tr>
<th>Graph</th>
<th>PA</th>
<th>ST</th>
<th>RE</th>
<th>CR</th>
<th>TT</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>23920</td>
<td>92253</td>
<td>2633</td>
<td>3129</td>
<td>12716</td>
<td>768</td>
</tr>
<tr>
<td>Patents</td>
<td>311</td>
<td>427</td>
<td>130</td>
<td>81</td>
<td>123</td>
<td>68</td>
</tr>
<tr>
<td>Mico</td>
<td>106</td>
<td>153</td>
<td>19</td>
<td>121</td>
<td>371</td>
<td>28</td>
</tr>
<tr>
<td>DBLP</td>
<td>10</td>
<td>22</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>Amazon</td>
<td>6</td>
<td>11</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Enron</td>
<td>40</td>
<td>84</td>
<td>4</td>
<td>5</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>24</td>
<td>37</td>
<td>4</td>
<td>7</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.14: Edge-induced Dynamic vs Static Speedup

4.7.3 Comparison with DAF and GSI

DAF and GSI are not able to do the vertex-induced matching. For fairness, we compare the performance of the edge-induced matching with DAF and GSI. DAF sets a threshold of the number of instances for a pattern in a data graph. The default value of that threshold is $10^4$. Table 4.5 presents the performance of our system once that threshold is reached.
Table 4.5: Performance with threshold (ms)

<table>
<thead>
<tr>
<th>Graph</th>
<th>PA</th>
<th>ST</th>
<th>RE</th>
<th>CR</th>
<th>TT</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>2.26</td>
<td>0.29</td>
<td>1.81</td>
<td>1.87</td>
<td>2.37</td>
<td>1.52</td>
</tr>
<tr>
<td>Patents</td>
<td>0.55</td>
<td>0.23</td>
<td>0.6</td>
<td>0.62</td>
<td>0.8</td>
<td>0.72</td>
</tr>
<tr>
<td>Mico</td>
<td>0.34</td>
<td>0.22</td>
<td>0.28</td>
<td>0.32</td>
<td>0.37</td>
<td>0.28</td>
</tr>
<tr>
<td>DBLP</td>
<td>0.22</td>
<td>0.28</td>
<td>0.24</td>
<td>0.26</td>
<td>0.23</td>
<td>0.24</td>
</tr>
<tr>
<td>Amazon</td>
<td>0.3</td>
<td>0.29</td>
<td>0.4</td>
<td>0.34</td>
<td>0.31</td>
<td>0.59</td>
</tr>
<tr>
<td>Enron</td>
<td>0.3</td>
<td>0.3</td>
<td>0.29</td>
<td>0.32</td>
<td>0.3</td>
<td>0.33</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>0.27</td>
<td>0.23</td>
<td>0.27</td>
<td>0.29</td>
<td>0.28</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Figure 4.15 shows the speedups of DGSM to DAF. DGSM is 108X faster than DAF on average. The minimum and maximum speedup is 32 and 527, respectively. DAF suffers the load imbalancing issue shown in Figure 4.7 because it maps each thread to a vertex in a candidate set.

As far as the unlabeled queries and data graphs are concerned, GSI cannot process many patterns in motif-4 even for a relatively small graph, like wiki-Vote, because of the memory limitation. GSI is able to answer clique-4 queries on most of the data graphs in the dataset except Mico and Livejournal. The query time of clique-4 on Patents, DBLP, Amazon, Enron, and wiki-Vote is 39.5, 153.3, 3.2, 23, and 20.6 in seconds, respectively. Figure 4.16 presents the speedups of DGSM with GSI. GSI can answer five queries out six in motif-4 on Amazon. The corresponding speedups are shown in Figure 4.17. Our performance is at least three orders of magnitude faster than GSI for motif-4 queries on Amazon.

We profiled GSI for these queries and found GSI launched 42 kernel calls on average ranging from 39 to 47 on Amazon. So many grid-level synchronizations and three memory allocations hurt its performance significantly. It is the reason why DGSM is much faster than GSI.

Besides patterns in motif-4, we also measured the performance of DGSM on dense queries, clique-5 through 10. The results are reported in Table 4.6 and Table 4.7. The cliques 8 through 10 take over a day to be accomplished on Livejournal and Mico. Therefore, we don’t report their performance here.
Figure 4.15: Speedup with DAF

Figure 4.16: Speedup of Clique 4 with GSI
4.7.4 Labeled Graph Performance

For labeled queries, we randomly assign a unique label out of ten labels to a vertex in a data graph in Table 4.1. We generate six sparse query patterns through random walks on the labeled data graphs. The number of vertices in the six queries ranges from four to nine,
and this number increases in order as the query index; that is, \(|V(Q_1)| = 4, |V(Q_2)| = 5,\) and so on. Table 4.8 and Table 4.9 show the performance of these six queries on the data graphs. The performance of Q6 on Livejournal and Mico is excluded because it takes more than one day to answer these two queries.

Table 4.8: Performance of 6 Labeled Patterns(ms)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patents</td>
<td>13.9</td>
<td>44.65</td>
<td>72.01</td>
<td>55.3</td>
<td>562.91</td>
<td>12908.8</td>
</tr>
<tr>
<td>DBLP</td>
<td>1.98</td>
<td>4.08</td>
<td>31.5</td>
<td>305.13</td>
<td>4425.91</td>
<td>46942.5</td>
</tr>
<tr>
<td>Amazon</td>
<td>0.68</td>
<td>0.6</td>
<td>1.3</td>
<td>1.8</td>
<td>0.8</td>
<td>1.72</td>
</tr>
<tr>
<td>Enron</td>
<td>0.94</td>
<td>7.43</td>
<td>58.95</td>
<td>125.03</td>
<td>727.63</td>
<td>22009.4</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>10.94</td>
<td>53.88</td>
<td>164.01</td>
<td>674.96</td>
<td>5500.78</td>
<td>98658.9</td>
</tr>
</tbody>
</table>

Table 4.9: Performance of 5 Labeled Patterns(ms)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>87.44</td>
<td>6460.88</td>
<td>51320.1</td>
<td>5988970</td>
<td>113077000</td>
</tr>
<tr>
<td>Mico</td>
<td>45.85</td>
<td>131.1</td>
<td>3370.69</td>
<td>67786.6</td>
<td>1521.49</td>
</tr>
</tbody>
</table>

Table 4.10 shows the performance of six labeled queries when the threshold is specified.

Table 4.10: Labeled Performance with Threshold(ms)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>0.42</td>
<td>0.96</td>
<td>3.92</td>
<td>1.1</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>Patents</td>
<td>1</td>
<td>0.58</td>
<td>0.84</td>
<td>5.4</td>
<td>5.27</td>
<td>5.28</td>
</tr>
<tr>
<td>Mico</td>
<td>0.39</td>
<td>0.43</td>
<td>0.44</td>
<td>0.44</td>
<td>0.5</td>
<td>0.49</td>
</tr>
<tr>
<td>DBLP</td>
<td>0.38</td>
<td>0.39</td>
<td>0.42</td>
<td>0.41</td>
<td>0.41</td>
<td>0.42</td>
</tr>
<tr>
<td>Amazon</td>
<td>0.35</td>
<td>0.4</td>
<td>0.39</td>
<td>0.4</td>
<td>0.4</td>
<td>0.46</td>
</tr>
<tr>
<td>Enron</td>
<td>0.12</td>
<td>0.12</td>
<td>0.13</td>
<td>0.13</td>
<td>0.14</td>
<td>0.13</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>0.08</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Figure 4.18 shows the speedups of these six queries to DAF. DAF cannot answer queries on Livejournal and Patents. Therefore, their speedups are not displayed in Figure 4.18. We achieve 57X speedup with DAF on average. The speedups range from 18X to 157X.
For GSI, no query can be done on Livejournal. The first two queries, $Q_1$ and $Q_2$, can be answered on most of the remaining data graphs. All the six queries can be answered on Amazon. The average speedup is 160X. The range is from 104X to 311X. Figure 4.19 shows the speedup on these six queries. The average speedup of these two queries is about 95X.

Figure 4.20 presents the speedups of the first two queries where the missing bar denote the query cannot be answered.

4.8 Conclusion

In this chapter, we introduced our new GPU-based subgraph matching system, DGSM, which is able to answer both vertex-induced and edge-induced queries. The DFS exploration grants our system the ability to handle large queries and data graphs. After integrating the shared memory and load imbalance optimizations, DGSM is further boosted. The results show that DGSM outperforms two state-of-the-art subgraph matching systems including both CPU-based and GPU-based frameworks. DGSM is about two or three orders of magnitude faster than DAF and GSI when answering unlabeled queries, and it is two orders of magnitude faster on labeled queries.
Figure 4.19: Speedup of Labeled Queries on Amazon with GSI

Figure 4.20: Speedup of Two Queries with GSI
5.1 Summary

In this thesis, we investigated accelerating two fundamental pillars in graph analytics, graph traversal and subgraph isomorphism, by using NVIDIA GPUs. We addressed several important performance issues in both domains through different strategies. Incorporating these strategies in our systems, our systems outperform the state-of-the-art counterparts as of the time of writing each system.

In the first part of this thesis, we discussed a new GPU-based edge-centric graph traversal system, named Graphie. Graphie adopts the edge-centric parallel model to maximally exploit data parallelism and achieve better load balance in graph traversal algorithms. Graphie also takes advantage of the special architecture features of modern NVIDIA GPUs including the Hyper-Q technique and shared memory to further improve the performance of graph traversal algorithms. We proposed two renaming processes in order to utilize the shared memory to reduce the latency of repeated memory accesses without sacrificing data parallelism. Unlike some frameworks which maintain an updated vertex array of size $N(V)$, we introduce a relatively small array to keep track of active partitions. This array not only guides the data movement from the host to the device, but also helps us to avoid unnecessary data movement. Furthermore, we employ the Hyper-Q technique to achieve grid level concurrency so that part of latency can be hidden by overlapping data transfers and kernel executions. The effectiveness of our approaches is validated by running three algorithms on seven real-world graphs consisting of both in-memory and out-of-memory graphs. On average, Graphie outperforms the state-of-the-art CPU-based and GPU-based graph traversal frameworks as of the time of writing.

In the rest of the thesis, we discuss our GPU-based pattern-aware subgraph matching system. The system consists of two stages. The first stage obtains the matching order of
query vertices based upon the topology of the query graph, and it is done on the host side. The query graph can be encoded as a series of set operations. This representation has the capability to avoid redundant computation in connectivity checks. Our system foregoes an automorphism test, while other systems either retain these duplicates throughout the matching or remove duplicates by implementing expensive automorphism tests on subpatterns. Including automorphism tests in a system will significantly hurt the performance because there is no efficient parallel automorphism test algorithm available for GPUs. In contrast to the above two methods, we enforce partial orders in each group of symmetrical vertices to remove redundant storage and computation originated from symmetries. The benefit of our approach is that our framework will not incur significant memory overhead nor wastes computation resources, and it reduces the performance because these duplicates are trimmed away by partial orders.

We adopt the DFS exploration strategy to cope with the exponential growth of the amount of subpatterns. First of all, this strategy limits the number of intermediate instances during the matching process. Second, this exploration strategy also eliminates communication among execution units and many grid level synchronizations when expanding subpatterns. They are two critical factors affecting the performance of a GPU system.

We design a GPU-friendly data structure, called VertexSet, to make device memory accesses coalesced. A coalesced memory access pattern is very important for the performance of GPU applications. Such a data structure not only facilitates the connectivity check in the iterative matching process, but also makes the backtracking easier. Throughout the matching process, threads in a warp work collectively to avoid warp divergence. Therefore, VertexSets will create a lot of pressure on registers because of this mapping approach. To mitigate register pressure, we cache the VertexSet metadata in the shared memory so that threads in a warp are able to access them for fast references. Leveraging the shared memory in this way guarantees data parallelism without sacrificing the performance since the metadata will be accessed millions of times throughout the
matching. However, the actual matched vertices are stored in a continuous chunk in the global memory. Therefore, the coalesced memory accesses are maintained. Furthermore, the VertexSet metadata help us to keep track of matching computations and remove redundant computations by moving them up as early as possible.

We deal with load imbalance issues by three measures: 1) reordering vertices by degrees, 2) edge-centric mapping, 3) dynamic scheduling. Our experimental results demonstrates that each one of three approaches attenuates the load imbalance to in subgraph matching to a certain degree. Combining all the techniques mentioned above, DGSM obtains at least two orders of magnitude speedup with DAF and GSI answering both labeled and unlabeled queries on seven data graphs.

5.2 Summary of Publications

This section is dedicated to my publications throughout my PhD study.


We studied fine-grained GPU spatial sharing among multiple applications. The goal of this study is to look for a resource partition such that the overall throughput is maximized and the QoS of one application is guaranteed. This effort led to a publication [24] in the 32nd Workshop on Languages and Compilers for Parallel Computing (LCPC, 2019). This work is not included in this thesis.

We have a paper in preparation based on the work in chapter 4.

I also made contributions to a co-authored paper that is not discussed herein. The paper is accepted by the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT, 2021).

5.3 Future Work

Graph analytics is a broad and evolving subject. It is impossible to cover all the aspects of graph analytics in a thesis or a book because it is a multidisciplinary topic. Hopefully, our work could pave the way for the future development and create a positive ripple effect
in this area. Although our work studies two major problems in graph analytics, it is still the tip of the iceberg. There are always open questions in graph analytics.

First, is there an efficient parallel DFS algorithm? As one of fundamental graph traversal strategies, DFS plays an important role in many graph algorithms. Its role is irreplaceable. However, it is challenging to implement a parallel version of DFS since there are strong dependencies in DFS.

Second, this thesis is focusing on a single GPU implementation. Our work can be further scaled both horizontally and vertically. We can leverage memory space on multiple GPUs in a single machine or a distributed environment to accommodate the memory need for large graphs. But it comes at the cost of communication and load imbalance. These are the interesting topics in the high-performance community.

Third, We concentrate on statics graphs in the thesis. In reality, graphs evolve frequently over time with relatively small changes. It would be expensive to restart an algorithm on the updated graphs. Therefore, people are craving for incremental algorithms. In contrast to batch algorithms which restart the computation on the update graph, incremental algorithms compute the updates caused by changes based on previous results. In general, incremental algorithms are less expensive than batch algorithms since the changes are small compared with original inputs. If parallel incremental algorithms are available, we expect to see a significant improvement on GPU-based graph frameworks.

Fourth, most graph algorithms are memory bound. They usually lead to GPU underutilization. How to increase GPU utilization when graph algorithms are running GPUs is an attractive direction for future study. GPU sharing is a promising candidate in such a context. How to partition resources between a graph application and another in a fine-grained way without degrading the performance of the graph application is an appealing research topic.
REFERENCES


algorithmic paradigm for parallel graph computations. In Proceedings of the 23rd
International Conference on Parallel Architectures and Compilation, PACT ’14, pp.
27–38.

[26] iNeumann, T. and G. Weikum (2010). The rdf-3x engine for scalable management of

mining system. In Proceedings of the Fifteenth European Conference on Computer
Systems, EuroSys 20, New York, NY, USA. Association for Computing Machinery.

International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3,

trade-offs in GPU implementations of graph algorithms. In 2016 IEEE International
Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May

presence of divergence with collaborative context collection. In Proceedings of the 48th

processing on gpus. In 2015 International Conference on Parallel Architecture and

graph processing on gpus. In Proceedings of the 23rd International Symposium on
High-performance Parallel and Distributed Computing, HPDC ’14, pp. 239–252.

graph processing method based on streaming topology to gpus. In Proceedings of the

[34] Kwak, H., C. Lee, H. Park, and S. Moon (2010). What is twitter, a social network or
a news media? In Proceedings of the 19th international conference on World wide web,
pp. 591–600. ACM.

computation on just a pc. In Proceedings of the 10th USENIX Conference on Operating
Systems Design and Implementation, OSDI’12, pp. 31–46.

[36] Lee, J., W. Han, R. Kasperovics, and J. Lee (2012). An in-depth comparison of
subgraph isomorphism algorithms in graph databases. Proc. VLDB Endow. 6(2),
133–144.

Proceedings of the International Conference for High Performance Computing,


