GRAPH PROCESSING ON GPU

ZHANG JINGBO

(B.E., UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that this thesis is my original work and it

has been written by me in its entirety.

I have duly acknowledged all the sources of information

which have been used in the thesis.

This thesis has also not been submitted for any degree in any

university previously.

Zhang Jingbo

Zhang Jingbo
July 17, 2013

Acknowledgment

I would like to express my greatest thank my PhD thesis committee members, Anthony K. H. Tung, Tan Kian-Lee and Sung Wing Ken, for their valuable time, suggestions and comments on my thesis.

I would like to express my deepest gratitude to my supervisor, Professor Anthony K. H. Tung, for his guidance, support and encouragement throughout my Ph.D. study. He has taught me a lot on research, work and life in the past five years, which will become my precious treasure in my life. Moreover, I am grateful for his generous financial support and tremendous mental assistance, especially when I was frustrated at times during the final stage of my Ph.D. study. His technical and editorial advice is essential to the completion of this thesis while his kindness and wisdom have made a great impact on my life. Professor Beng Chin Ooi deserves my special appreciation. He is the greatest figure I have met in my life. As a visionary leader of our database group, he acts as a passionate doer, an earnest advisor and a considerate friend.

My sincere thanks also go to Dr. Wang Nan. Dr. Wang provided me resources to start my ventures on graph mining, and her insights on graph mining and encouragement were of great help for my research. I am also indebted to Dr. Seth Norman Hetu. Apart from contributing helpful discussions to refine my work, he spent much effort in updating my writings. My senior Dr. Xiang Shili taught and encouraged me a lot of things. Dr. Zhu Linhong, Dr. Wu Min and Myat Aye Nyein, who are my closest friends, accompanied, discussed, and supported me in the past years.

The last seven years in National University of Singapore have become a wonderful journey in my life. It is my great honor to be a member of our database group, a big family full of joy and research spirit. I am very thankful to our iData group members (including previous and current members). They are Yueguo Chen, Bingtian Dai, Wei Kang, Chen Liu, Meiyu Lu, Zhan Su, Nan Wang, Xiaoli Wang, Shanshan Ying, Feng Zhao, Dongxiang Zhang, Zhenjie Zhang, Yuxin Zheng, Jingbo Zhou. Besides, it is my great pleasure to work together with our strong team of NUS Database Group, including Zhifeng Bao, Ruichu Cai, Yu Cao, Su Chen, Ming Gao, Bin Liu, Xuan Liu, Wei Lu, Weiwei Hu, Mei Hui, Feng Li, Yuting Lin, Peng Lu, Wei Pan, Yanyan Shen, Lei Shi, Yang Sun, Jinbao Wang, Huayu Wu, Ji Wu, Sai Wu, Hoang Tam Vo, Jia Xu, Liang Xu, Xiaoyan Yang, and Meihui Zhang. Throughout the long period of PhD study, we discuss and debate about research problems, work together and collaborate in the projects, encourage and care for each other, and entertain as well as do sports together.

I am grateful to my parents, Shuming Zhang and Yumei Lin, for their dedicated love, care and the powerful and faithful support during my studies. Their nutrition and patience have brought me infinite energy to go through all the thorns and tribulations. My deepest love is reserved for my wife, Lilin Chen, for her unconditional support and encouragement during the past two years.

Finally, I also want to thank NUS for providing me the scholarship so that I can concentrate on the study.

Contents

Summary

Graph mining and data management has become a significant area because more and more new applications to various data mining problems in social networking, computational biology, chemical data analysis and drug discovery are emerging recently. Although traditional mining methods have been extended to process graphs, many graph applications still confront huge challenges due to continuous and overwhelming edges to be processed with limited resources. Social networks, web graphs and protein interaction graphs are difficult to handle because they cannot be easily decomposed into small parts that could be further processed in parallel. As graphs grow larger and larger, new processing techniques with higher computing power are demanded for mining massive graphs. Designing scalable systems for analyzing, processing and mining huge real-world graphs has also become one of the most emerging problems.

The research in this thesis has explored and utilized the state-of-the-art GPGPU techniques over large graph mining. By understanding the limitations of heterogeneous hardware, triangulation, as a representative of graph mining algorithms, was implemented to be accelerated by many-core GPUs in Chapter 3. Associated graph data structures and blended algorithm structures were designed in this chapter as well. This is the first and successful attempt to accelerate graph triangulation using GPGPU techniques. Afterwards, a synchronous iterative GPU-accelerated graph processing model was abstracted and proposed in Chapter 4. A generic system (SIGPS) was then implemented based

on this model. Specifically, a vertex API was provided for users who want to design their own algorithms with the assistance of a functional library of mining algorithms. Together with the vertex API and algorithm library, several system supporting modules marked off the system hierarchy. This system could bring an impressive impact over the graph mining community since it provided a systematic solution for implementing efficient graph mining algorithms on GPU-accelerated computing platforms. Moreover, in order to further enhance the system performance, an asynchronous disk-based model was then designed to support asynchronous computing over GPUs in Chapter 5. A novel parallel sliding windows method was employed on GPU memory. Two newer operational APIs named “sync” and “update” replaced the vertex API. Asynchronous-SIGPS (ASIGPS) could be used to execute several advanced data mining, graph mining, and machine learning algorithms on very large graphs.

It is noted that there may be a few problematic issues involved in the system since designing effective and efficient systems across heterogeneous platform is complicated. As a potential solution for large scale domain applications on personal computers, more graph mining algorithms need to be implemented to constitute the library of the system and more efforts need to be paid to solve all the problems related to the implementation of the hybrid system.

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we will describe the background of computing and graph mining, give a general overview of the state-of-the-art GPGPU techniques in the current literature, and present the rationale of our study on utilizing GPU to accelerate mining over large graphs.

1.1 Background

One of the major changes in the computer software industry has been the move from serial programming to parallel programming. The graphics processor unit (GPU) by its very nature is the device designed for high-speed graphics present in most modern PCs, which are inherently parallel. The state-of-the-art GPGPU techniques take a simple model of data parallelism and incorporate it into a programming model without the need for graphics primitives. On the other hand, the ability to mine data to extract useful knowledge has become one of the most important challenges in government, industry, and scientific communities. In most domains, there is a lot of interesting knowledge that can be mined out of relationships between entities.

1.1.1 Supercomputing and Desktop-computing with GPUs

Supercomputers are typically at the leading edge of the technology curve. In 2010, the annual International Supercomputer Conference in Hamburg, Germany, announced that a NVIDIA GPU-based machine had been listed as the second most powerful computer in the world, according to the top 500 list (http://www.top500.org). In 2011, NVIDIA CUDA-powered GPUs grasped the title of the fastest supercomputer in the world. It was suddenly noticeable to everyone that GPUs had arrived in a very big way on the high-performance computing landscape, as well as the humble desktop PC.

Supercomputing is the driver of many of the technologies we see in modern-day processors. Due to the need for ever-faster processors to process ever-larger datasets, the industry produces ever-faster computers. It is through some of these evolutions that GPGPU technology has come about today.

Both supercomputers and desktop computing are moving toward a heterogeneous computing route –that is, they are trying to achieve performance with a mix of CPU and GPU technology. Jaguar, the fastest supercomputer, code-named Titan, has almost 300,000 CPU cores and up to 18,000 GPU boards to achieve between 10 and 20 petaflops per second of performance. People can now put together or purchase a desktop supercomputer with several teraflops of performance. This would have given the first place in the top 500 list [1] at the beginning of 2000, which is just 13 years ago.

1.1.2 Graph Processing and Mining

Graphs are regarded as one of the most ubiquitous models of both natural and human-made structures. A lot of practical problems in scientific and engineering areas can be modeled by graphical model. As a very popular and flexible data abstraction for connected entities, graphs capture the relationship among these entities. For example,

[1]IBM ASCI Red with 9632 Pentium processors

social networks, popularized by Web 2.0, are graphs that describe relationships among people. Well defined graph theory can be applied to processing the graph and return interesting results. With the increasing demand on the analysis of large amounts of structured data, graph processing has become an active and important theme in data mining. On one side, growing richer information potentially extracted from large graphs has triggered progressively more sophisticated analysis of graph data. On the other side, since dense graph pattern captures more internal connections within a graph, researchers from various fields are all using dense subgraphs to understand complex systems better.

Dense subgraph mining is close-relative but simpler when comparing with the traditional clustering which requires a strict partitioning of the graph. Exact mining methods are usually time consuming algorithms, some of which are even regarded as NP-hard problems. People then opt for some more time efficient solutions. This type of algorithms can be categorized into three groups, namely enumeration, fast heuristic enumeration and bounded approximation.

1.1.3 General Purpose Computation on GPU

Graphics processing units (GPUs) are devices present in most modern PCs. They provide a number of basic operations to the CPU, such as rendering an image in memory and then displaying that image onto the screen. A GPU will typically process a complex set of polygons, a map of the scene to be rendered. It then applies textures to the polygons and then performs shading and lighting calculations.

General-Purpose computation on Graphics Processing Units (GPGPU) is a technique of using GPU to perform computation in applications traditionally handled by CPU. After shifting from fast single instruction pipeline to multiple instruction pipelines, modern computer systems have evolved into multiple threads architecture in the coming era of Tera-scale Computing. Dual-core and many-core facilities have greatly improved the executing performance without impacting thermal and power delivery. Moreover, some special-purpose devices are designed for accelerating the data processing, such as ASIC, FPGA and GPU. As a special-purpose co-processor to CPU, a graphics processing unit (GPU) was originally designed for accelerating graphics rendering operations. In the last decade, modern GPUs have evolved to be many-core processors with the potential of high parallelism. They have displayed an impressive computational capability as well as higher memory bandwidth compared to CPUs. Actually, general purpose computing has arisen to exploit the potential computing power from systems equipped with graphics cards. More and more developers have moved the computationally intensive parts of their applications to GPUs for acceleration. There are currently many GPU-accelerated applications and the list grows monthly. NVIDIA showcases many of these on its community website at http: //www.nvidia.com/object/cuda_apps_flash_new.html. Considering the performance-to-price ratio (cost-utility), the possibility of releasing the potential power of general computer system has become an attractive alternative option to traditional distributed supercomputer systems.

1.1.4 Graph Processing on GPU

For the past decade, various graph mining techniques have been developed to discover patterns, clusters, and classifications from various kinds of graphs. Many algorithms focus on the effectiveness of mining, while other researches aim at the performance improvement of the specific methods. Utilizing parallel architectures has been a viable means to improving graph processing performance. Modern GPUs have displayed an impressive computational power as well as higher memory bandwidth compared to CPUs. Given the success of GPGPU in many areas of scientific computing, graph processing on GPU appears to be necessary to overcome the resource limitations of single processors. A GPU can be regarded as a massively multi-threaded many-core processor. Its cores are designed to be virtualized, and its threads are managed by the hardware, which simplifies GPU programs and improves algorithm scalability and portability. By taking advantage of the massive computation power and the high memory bandwidth, GPUs can be used by many graph (mining) applications as an accelerator to compute-intensive algorithms. To process excessive graph data with limited resources, researchers combine graph mining with the state-of-the-art GPGPU techniques. Moreover, energy efficiency improvement while the system provides an order of magnitude increase in computational power is another vital factor to process graphs on GPU.

1.1.5 Graph Processing System

In order to achieve efficient and effective graph data processing on GPU, the implementation of existing graph processing algorithms on GPU and a generic graph processing system are two important research issues. For the first issue, as is well known, most graph processing algorithms are designed to be sequential and memory bounded. How to parallelize graph processing algorithms effectively and bypass the memory restriction successfully are challenging problems to be solved. For the other issue, Internet companies have created scalable infrastructure. One example is that google has been using a distributed high performance graph processing system named Pregel to process its massive graph data. Pregel can easily scale to billions of vertices and edges on google's distributed many-core-CPU system. The applicability and usability of Pregel are pretty impressive. Mining huge graphs on general computer systems, however, is still a challenge. On one hand, general computer systems are equipped with fewer computing cores than traditional supercomputers. Hundreds of thousands of vertices and millions of connections among vertices make traditional graph mining operators a huge burden for a normal computer. Close-clique detection, for example, has been proven to be an NP-Complete problem. Even the running time of heuristic algorithms or approximation algorithms on such large graphs have exceeded the tolerance of human beings. On the other hand, limited memory is another prohibitive factor for the scalability of high performance computing on general computers. A large graph cannot even be loaded into memory for any further processing. Therefore, a generic graph processing system implemented on general computers equipped with GPUs is preferable to the data mining community.

1.2 Research Gaps, Purpose and Contributions

As graphs grow incredibly large in size, many graph applications encounter great difficulties due to the insufficiencies of computing power and the limitations of computing platforms. Since GPU provides potential opportunities of highly parallel computing, the question of how to apply the state-of-the-art GPGPU techniques over massive graph applications has become a huge challenge. Research gaps for the current application of GPGPU over large graphs are summarized below:

1. Although traditional mining methods can be utilized to process large graphs, they are highly constrained when the system resources are limited. When GPU is employed to accelerate graph algorithms, whether and how the traditional mining methods can be extended to parallelized version by way of GPGPU techniques is still problematic.

2. There are some existing graph processing systems that incorporate a library of graph mining algorithms. However, some of these libraries are only applicable to small graphs while others are only designed for processing large graphs in distributed environments. Moreover, most existent graph processing systems only provide naive APIs for invoking existing routines that implement classic mining

algorithms. It is difficult for users to design their own algorithms, which are usually more complicated.

3. Currently, most graph processing systems support parallel graph mining algorithms. Nevertheless, none of them provide algorithms utilizing GPGPU techniques that can take advantage of the potential high performance computing power from modern GPUs.

4. Most generic parallel systems are based on Bulk Synchronous Parallel model that trades off performance for simplicity in algorithm design. There are limited solutions that can support asynchronous processing.

The main aim of my research was to utilize GPGPU techniques over large graph mining. By understanding the limitations of heterogeneous hardware, I designed graph mining algorithms on GPU. In order to provide a systematic solution for implementing efficient graph mining algorithms, I proposed a synchronous GPU graph processing model and implemented a generic graph processing system over GPU-accelerated general computers. The specific objectives of this study were to:

1. design GPU-accelerated mining algorithms over large graphs. We initially designed a triangulation operator over GPU. We then summarized the associated graph data structures and the blended algorithm structure design from graph processing algorithms such as SSSP and PageRank.

2. propose a synchronous graph processing model over GPU-accelerated platform. By simplifying the blended algorithm structure, we presented a graph processing model that is based on bulk synchronous parallel computing. A generic vertex API was proposed to assist algorithm design.

3. design and implement a generic graph processing system that employs the synchronous graph processing model. A real graph processing system over heterogeneous platform was implemented in C++ and CUDA. The vertex API, graph processing library, and system supporting modules have differentiated the hierarchy of the system.

4. investigate the limitation of synchronous model and design an asynchronous one. By fully studying the limitation of our synchronous model, an improved model that provided asynchronous computing was then designed. The vertex API was then replaced by two new operational APIs named “sync” and “update” respectively.

5. design and implement a generic graph processing system that supports the asynchronous processing over GPU-accelerated large graph applications. We would then redesign the graph processing system on top of the asynchronous graph processing model with better system modularity.

The comprehensive experimental results of this study may have a significant impact on both successfully applying GPGPU techniques to speed up large graph applications with limited resources and providing systematic generic graph mining solutions.

To design an effective and efficient system accelerated by GPU is complicated since it contains a lot of new research issues that are related to the library building, system design and hardware tuning. There may be a few problematic issues involved. It is also understood that we only focus on graph processing on top of general computer systems. More data mining applications and graph processing accelerated by connected distributed GPU nodes are very interesting but beyond the scope of this thesis.

1.3 Thesis Organization

Hereby, we outline the organization of this thesis. The rest of the thesis contains 5 chapters.

Chapter 2 consists of two main sections. The fist section is the background and related work of graph mining on GPU. The second section introduces the mining of DN-Graph, which directly led to the research of this thesis.

Chapter 3 presents our solution of accelerating a dense sub-graph mining operator on GPU. Since memory and computing power are main bottlenecks of the graph mining system, we utilize a streaming approach to partition the graph and take advantage of the state-of-the-art GPGPU techniques for bounding acceleration. A two-level triangulation algorithm is employed to iteratively drive triangulation operator on GPU. In addition, several novel GPU graph data structures are proposed to enhance graph processing efficiency and data transfer bandwidth.

We then extend our work on accelerating graph mining operators in a systematic solution in Chapter 4. An iterative graph processing model on GPU-accelerated platform is proposed. Based on this model, a generic system equipped with a set of easy-to-extend Vertex APIs is then implemented over the model. Automatic parallelization and GPU execution configuration are provided in the system. Emulating shared memory model is also designed for vertex communication.

In Chapter 5, we optimize the graph processing model to support asynchronous processing on GPU. After system re-design, the “Asigps” has better modularity and encapsulation. An improved new set of easy-to-extend Vertex APIs are designed, so that users have higher degree of freedom to design their own algorithms. Asigps is a disk-based GPU-accelerated system for computing efficiently on graphs with billions of edges. A novel parallel sliding windows method was implemented on GPU memory. Asigps is designed to support several advanced data mining graph mining, and machine learning algorithms on very large graphs using just a single GPU-accelerated personal computer.

Finally, Chapter 6 concludes this thesis and discusses some directions for future work.

Chapter 2

Background and Related Works

In this chapter, we first introduce preliminaries and some fundamental graph structures, which are employed in our proposed system or some closely related works. Then, we focus on the work that led to this thesis. More specifically, we first present some definitions of notations and discuss some system metrics in the related works. Then we review the GPGPU background and graph processing on GPU in the literature. Last but not least, we introduce our DN-Graph mining work that induces the demand and the subsequent research in this thesis.

2.1 Preliminaries

2.1.1 Graph Notations and Definitions

Let $G = (V, E)$ be defined as an undirected simple graph with a set of nodes V and a set of edges E. A dense graph pattern [1] is a connected subgraph $S = (V', E') \subset G$ and $V' \subset V, E' \subset E$, which has significant more internal connections with respect to the surrounding vertices.

[1]or dense subgraph

A triangle $\triangle = (V_\triangle, E_\triangle)$ of the graph G is also defined as a three node subgraph with $V_\triangle = \{u, v, w\} \subset V$ and $E_\triangle = \{(u,v),(u,w),(v,w)\} \subset E$. We use the symbol $\delta(G)$ to denote the number of triangles in graph G. Additionally, we employ the symbol $\delta(u)$ to denote the number of the triangles the vertex u participates in and the symbol $\delta(u,v)$ to denote the number of triangles the edge (u,v) is involved in.

2.1.2 Graph Memory Assumptions

Informally, we assume a personal computer system is equipped with limited memory (DRAM) capacity. The graph structure, edge values and vertex values do not fit into memory. On the contrary, the edges or values associated to any single vertex can be stored in the memory.

Assumption 2.1.1. COMPUTATIONAL CONSTRAINTS

1. *We assume the amount of memory to be only a small fraction of the memory required for storing the complete graph.*
2. *We assume there is enough memory to contain the edges and values associated to any single vertex in the graph.*

2.1.3 Heterogeneous System Metrics

Almost all processors work on the basis of the process developed by Von Neumann, in which approach, the processor fetches instructions from memory, decodes, and then executes that instruction. As is described in DEFINITION 2.1.1, a stored-program digital computer is one that keeps its programmed instructions, as well as its data, in read-write, random-access memory (RAM). The principle of locality is one of the most important characters of modern computer systems. As is defined in DEFINITION 2.1.2, modern programs tend to reuse data and instructions they have accessed recently.

Definition 2.1.1. VON NEUMANN ARCHITECTURE

The Von Neumann architecture describes a design architecture for an electronic digital computer with subdivisions of a processing unit consisting of an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms.

Definition 2.1.2. THE PRINCIPLE OF LOCALITY

Programs access a relatively small portion of the address space at any instant of time.

To evaluate the performance of a system, processor and memory frequency, communication bandwidth, and the system data throughput are basic metrics. As is defined in DEFINITION 2.1.3, bandwidth refers to the maximum amount (capacity) of data that can pass through the communication channels per second. A modern processor typically runs at a high frequency in speed [2]. A modern DDR-3 memory, which is paired with standard processors, can run at a comparable frequency [3]. The ratio of clock speed to memory is an important limiter for both CPU and GPU throughput, which is defined in DEFINITION 2.1.4.

Definition 2.1.3. BANDWIDTH

Bandwidth is a measurement of bit-rate of available or consumed data communication resources expressed in bits per second or multiples of it. In practice, the digital data rate limit (or channel capacity) of a physical communication link is proportional to its bandwidth in hertz.

Definition 2.1.4. THROUGHPUT

Throughput is the average rate of successful message delivery over a communication channel. The data may be delivered over a physical or logical link, or pass through a certain network node. The throughput is usually measured in bits per second (bit/s or bps).

In heterogeneous systems, there are more than one types of processors. For example, our personal computer systems are equipped with multi-core CPU and many-core GPU processors. Applications designed for such hybrid system have adjustable parameters for different types of

[2] 4 GHz
[3] around 2 GHz

computing modes. The host mode is defined to be the state in which an application is only executed by CPU without any assistance of other co-processors. The device mode is defined to be the state in which an application is executed by co-processors, such as GPU or FPGA. The hybrid mode is defined to be the state in which an application is executed by both CPU and GPU.

To quantify the efficiency and performance of an application running on heterogeneous system, researchers usually employ the speedup and efficiency metrics. Intuitively, the speedup of a parallel code refers to how much faster it runs than a corresponding sequential algorithm does. The efficiency is a measure of the fraction that the available processing power is being used. According to the computing modes the application is in, the speedup and efficiency can be defined formally as follows:

Definition 2.1.5. SPEEDUP

The speedup of a parallel algorithm is defined to be the ratio of the rate at which when it is run on N processors to the rate at which it is processed by just one. Technically, if T_1 and T_N are the time required to complete some job on 1 and N processors respectively, the speedup S can be defined as follows:

$$S = \frac{T_1}{T_N} \tag{2.1}$$

In order to evaluate the performance of a parallel algorithm, there are different ways to compute the speedup, according to the structure of the algorithm. For example, in parallelized triangulation, if $T_1(\Delta(G))$ and $T_N(\Delta(G))$ are the time required to employ triangulation over Graph G on 1 and N processors respectively, global speedup can be defined as S_g in the following fomula; if $T_1(\lambda(e))$ and $T_N(\lambda(e))$ are the time required to employ triangulation over an edge e on 1 and N processors respectively, local speedup can be defined as S_l in the following fomula as well:

- GLOBAL SPEEDUP: $S_g = \frac{T_1(\Delta(G))}{T_N(\Delta(G))}$
- LOCAL SPEEDUP: $S_l = \frac{T_1(\lambda(e))}{T_N(\lambda(e))}$

Definition 2.1.6. EFFICIENCY

The efficiency of a parallel algorithm is defined to be the effectiveness of parallel algorithm relative to its sequential counterpart. Simply put, it is the speedup per processor. Technically, let N be the number of processors in the parallel environment, efficiency E is defined in terms of the ratio of the sequential cost C_1 to the parallel cost C_N.

$$E = \frac{C_1}{C_N} = \frac{T_1}{N \times T_N} \tag{2.2}$$

we also define global efficiency E_g and local efficiency E_l as follows:

- GLOBAL EFFICIENCY: $E_g = \frac{T_1(\Delta(G))}{N \times T_N(\Delta(G))}$
- LOCAL EFFICIENCY: $E_l = \frac{T_1(\lambda(e))}{N \times T_N(\lambda(e))}$

2.2 GPGPU Background

2.2.1 Parallel Programming Model

Many parallel programming languages and models have been proposed in the past several decades [35]. The Message Passing Interface (MPI) is widely used for distributed computing environment while OpenMP^{TM} is the de facto standard for shared-memory multi-core CPU systems. CUDA[4] is the GPGPU programming model proposed by NVIDIA Corporation [1]. Compared to the low scalability and weak thread management of multi-core CPU environment, CUDA provides a higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.

Actually, CUDA programming model employs SPMD (Single Program Multiple Data) manner when running on GPU. Compared with threads in CPU, threads in GPU is lightweight, which can be scheduled with extremely low cost [25]. Additionally, CUDA has a hierarchy of memory architecture. Analog to main memory, GPU global memory is off-chip memory that has the largest size but cost the most when being accessed. Constant memory and texture memory has caches and specific usage for higher performance. On-chip shared memory, analog to the CPU caches, and hundreds of registers can be accessed in the fastest speed but they are also limited in size on graphics chip. Threads are organized in units named “warp”, which can access consecutive memory locations with minimum cost [41]. The bottleneck of CUDA programs is usually found to be the high-speed PCI-Express bus that transfers data from main memory to GPU memory.

2.2.2 GPU Cluster Layout

Cluster computing became popular in 1990s along with ever-increasing clock rates. A general cluster consists of a number of commodity PCs bought or made from off-the-shelf parts and connected to an off-the-shelf 8-, 16-, 24, or 32-port Ethernet switch. Used together, the combined power of many machines hugely outperformed any single machine with a similar budget.

[4]Compute Unified Device Architecture

GPU computing today, as a disruptive technology that is changing the face of computing, is just like cluster computing. Combined with the ever-increasing single-core clock speeds it provides a cheap way to achieve parallel processing. The architecture inside a modern GPU is no different from a cluster. As is illustrated in Figure 2.1, there are a number of streaming multiprocessors (SMs) that are akin to CPU cores. These are connected to a shared memory/L1 cache. This is connected to an L2 cache that acts as an inter-SM switch. Data can be held in global memory storage where it is then extracted and used by the host, or sent via the PCI-E switch directly to the memory on another GPU. The PCI-E switch is many times faster than any networks's interconnect. The node may itself be replicated many times, as is shown in Figure 2.1. This replication within a controlled environment forms a cluster.

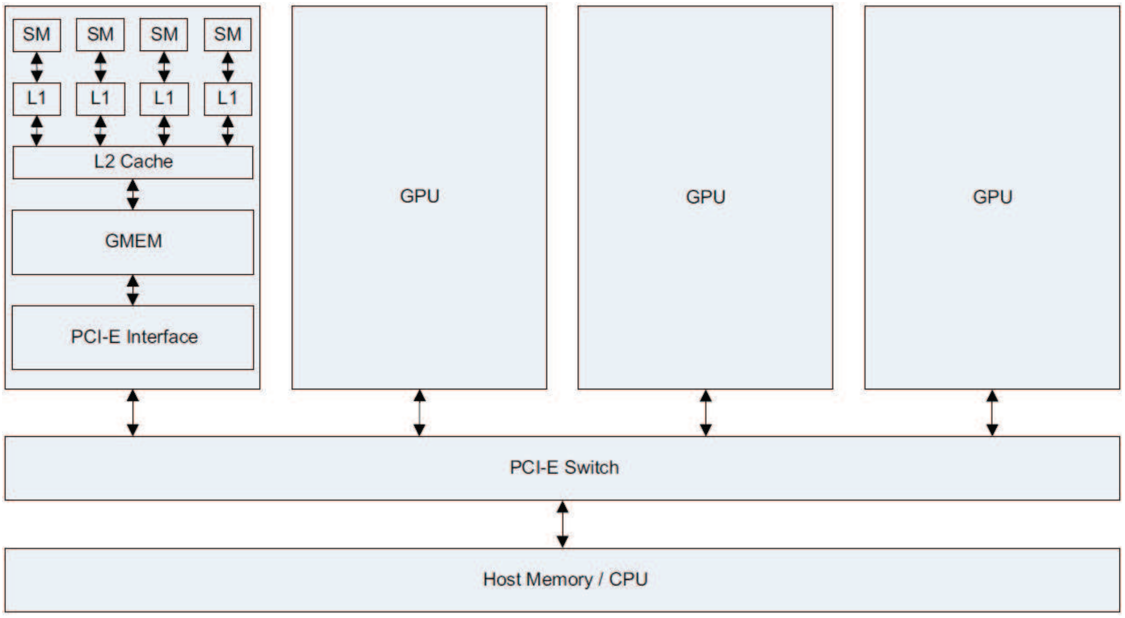
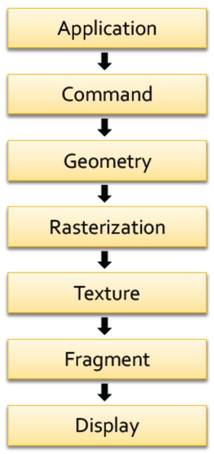


Figure 2.1: GPUs Cluster Layout

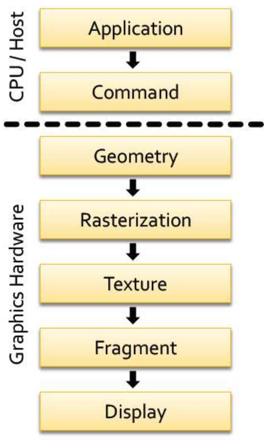
2.2.3 GPU Evolution

Graphics chips started as fixed function graphics pipelines. Over the years, these graphics chips became increasingly programmable, which led NVIDIA to introduce the first GPU or Graphics Processing Unit. In the 1999-2000 timeframe, computer scientists in particular, along with researchers in fields such as medical imaging and electromagnetics started using GPUs for running general purpose computational applications. They found the excellent floating point performance in GPUs led to a huge performance boost for a range of scientific applications. To use graphics chips, programmers had to use the equivalent of graphic API to access the processor cores. This was the advent of the movement called GPGPU or General Purpose computing on GPUs.

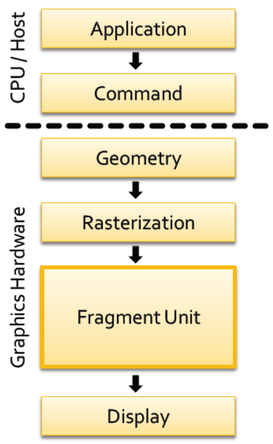
However, the difficulty of using graphics programming languages to program the GPU chips has limited the accessibility of tremendous performance of GPUs. Developers had to make their scientific applications look like graphics applications (use graphics APIs) and map them into problems that drew triangles and polygons. This limitation makes only a few people can master the skills which are necessary to use these chips to achieve performance. One of the important steps was the development of programmable shaders. These were effectively little programs that the GPU ran to calculate different effects. The rendering was no longer fixed in the GPU; through downloadable shaders, it could be manipulated. This was the first evolution of general purpose graphical processor unit (GPGPU) programming, in that design had taken its first steps in moving away from fixed function units. Then a few brave researchers made use of GPU technology to try and speed up general-purpose computing. This led to the development of a number of initiatives (e.g., BrookGPU [11] , Cg [34], CTM [6], etc.), all of which were aimed at making the GPU a real programmable device in the same way as the CPU. In order to exploit the potential power and bring this performance to the larger scientific community, NVIDIA devotes into modifying the GPU to make it fully programmable for scientific applications and adding support for high-level languages like C and C++. This led to the CUDA architecture for the GPU.



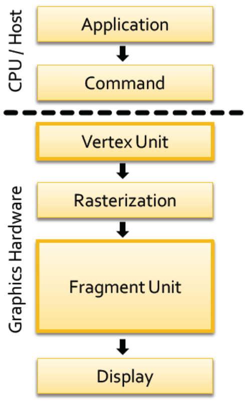
(a) Traditional Model



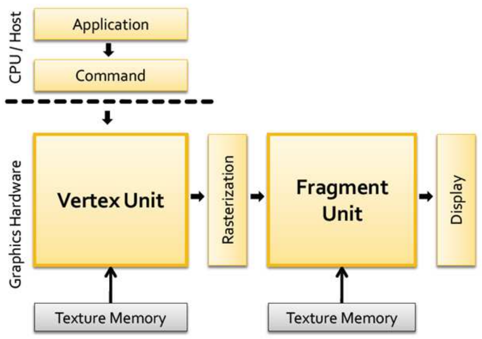
(b) A Dedicated Hardware



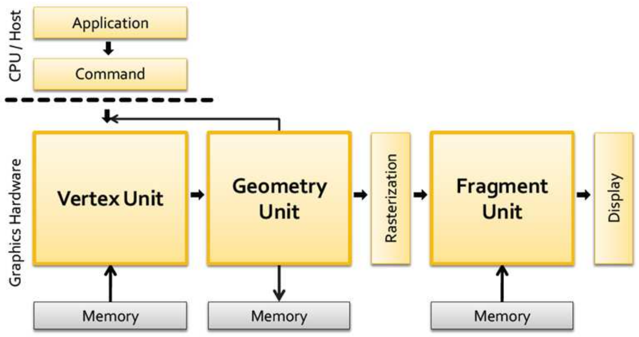
(c) Graphics Pipeline in 2000



(d) Graphics Pipeline in 2001-2002



(e) Graphics Pipeline in 2003



(f) Graphics Pipeline in 2007

Figure 2.2: Graphics Pipeline Evolution

Figure 2.2 shows the graphics pipeline evolution history. More specifically, Figure 2.2(a) describes the traditional model for 3-D rendering, in which there are 7 main stages in the graphics pipeline. The input of this referring model includes vertices and primitives, transformation operators, lighting parameters and so forth. The output of the model is a 2D image for display. The application stage describes the application program running on the CPU, example of which probably consists of simulation, input event handles, modify data structure, database traversal, primitive generation and utility functions. The command stage feeds commands to the graphics subsystem. In this stage, commands are buffered before being interpreted, data input are unpacked and converted into a suitable format while graphics state is maintained. The geometry stage mainly applies per-polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping which may be hardware-accelerated. Instead of the per-polygon operations in the geometry stage, the rasterization stage has per-pixel operations. Rasterization is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format. Operations of the rasterization stage include the simple operation of writing color values into the frame buffer, or more complex operations like depth buffering, alpha blending, and texture mapping, which may be hardware accelerated. In computer graphics, texture is a bitmap image applied to a surface in computer graphics. Texture mapping is a method for adding detail, surface texture, or color to a computer-generated graphic or 3D model. Similarly in the texture stage, texture filtering, which is also called as texture smoothing from other view, is the method used to determine the texture color for a texture mapped pixel, using the colors of nearby texels (pixels of the texture).

Starting from Figure 2.2(c), texture and fragment stage were combined to form a new stage named fragment unit, which became more programmable (via assembly language) in year 2000. This year memory in this programmable stage was read via “dependant” texture lookups, program size was limited and no real branching and looping were supported. Figure 2.2(d) shows in 2001 geometry stage became programmable (still via assembly language) and was called vertex unit. There were no memory reads supported in this stage and program size was still limited as well as the same situation of branching and looping compared to 2000. Then things improved in 2002 so that vertex unit can do memory reads and the supported maximum program size was increased and branch as well as some higher level languages such as HLSL and Cg were supported. However, both the vertex and fragment units could not write to memory but frame buffer. And there were no integer math and bitwise operators. In 2003, GPUs became mostly programmable. Although still inefficient, in Figure 2.2(e), "multi-pass" algorithms allowed writes to memory [5] [6]. Finally, as illustrated in Figure 2.2(f), processing units were "unified" so that the new geometry unit that operates on a primitive can write back to memory.

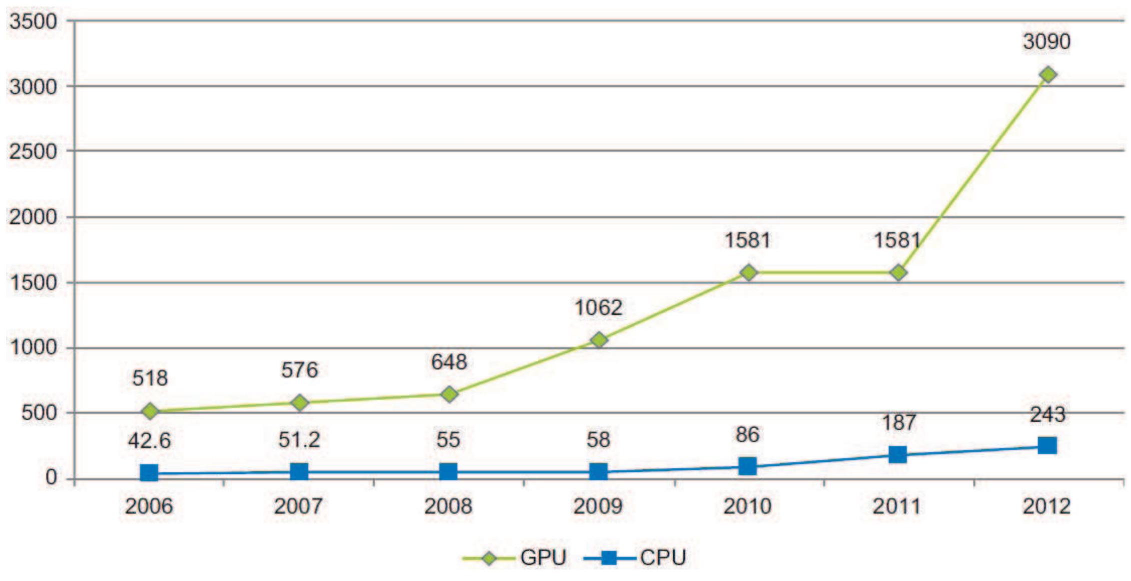


Figure 2.3: CPU vs GPU in Peak Performance (gigaflops)

2.2.4 CPU vs GPU

CPUs and GPUs are architecturally very different devices. CPUs are designed for running a small number of potentially quite complex tasks while GPUs are designed for running a large number of quite simple tasks.

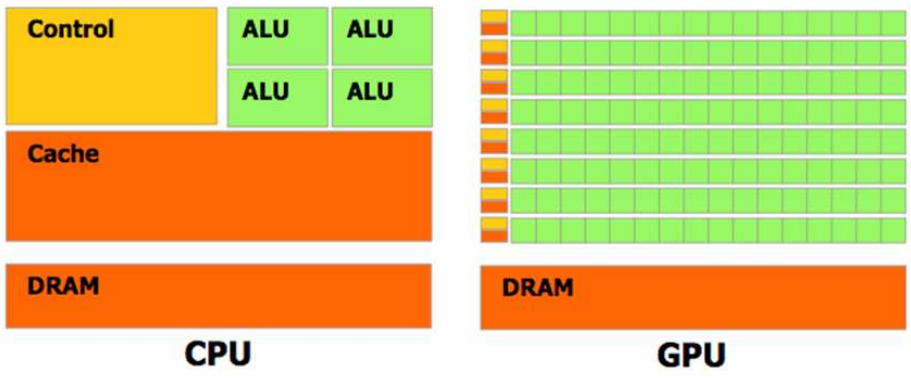
If we look at the relative computational power in GPUs and CPUs, we get an interesting graph (Figure 2.3). We start to see a divergence of CPU and GPU computational power until 2009 when we see the GPU finally break the 1000 gigaflops or 1 teraflop barrier. At this point

[5] write to the frame buffer in the first pass

[6] the frame buffer is re-bound as a texture and is read in the second pass

of time, the GPU hardware is moving from the G80 [7] to the G200 [8] and then to the Fermi [9] evolution. This is driven by the introduction of massively parallel hardware.

In Figure 2.3 we can also observe that NVIDIA GPUs make a leap of 300 gigaflops from the G200 architecture to the Fermi architecture, nearly a 30% improvement in throughput. By comparison, Intel's leap from their core 2 architecture to the Nehalem architecture sees only a minor improvement. Only with the change to Sandy Bridge architecture do we see significant leaps in CPU performance. The traditional CPUs are aimed and good at serial program execution while the GPUs are designed to achieve their peak performance only when fully utilized in a parallel manner.



(a) (b)

Figure 2.4: CPU vs GPU

There is a discrepancy in floating-point capability between the CPU and the GPU. GPU is specialized for compute-intensive, highly parallel computation. Therefore, more transistors are devoted to data processing rather than data caching and flow control in GPU. Figure 2.4 schematically illustrates these differences between the design of CPU and GPU.

CPU and GPU have different thread environment. The CPU has a small number of registers for each core, which must be used to execute any given task. To achieve this, CPU cores need to perform fast but expensive context switch among tasks. In contrast, instead of having a single

[7] 128 CUDA core device
[8] 256 CUDA core device
[9] 512 CUDA core device

set of registers, GPU cores have multiple banks of registers. A context switch of GPU threads simply involves setting a bank selector to swap in/out the current set of registers, which is much faster than saving to off-chip global memory.

Definition 2.2.1. SPATIAL LOCALITY

Data that is close to the last accessed data will likely be accessed in the future.

Definition 2.2.2. TEMPORAL LOCALITY

Data that has been accessed before, will likely be accessed again.

Another difference between CPU and GPU is about the principle of locality, which is defined in Definition 2.1.2. More specifically, spatial locality (Definition 2.2.1) and temporal locality (Definition 2.2.2) are two types of locality to be considered by programmers for a computer system. CPU is designed to run software where the programmer does not have to care about locality. On the contrary, GPU is designed with granting programmers the freedom of dealing with locality. The simple process of planning ahead allows the programmer to schedule data loads into the on-chip memory before they are needed.

One more important distinction between GPU and CPU is cache coherency. Although GPUs of early generation have no general memory cache, more and more new-born ones are equipped with hierarchical caches. For instance, the new Fermi and Kepler GPUs [10] have a different cache coherent mechanism from a general cache-coherent system. Specifically, a write to a main memory location needs to be communicated to all levels of cache in all cores. Thus, all CPU cores see the same view of memory at any point in time. This is one of the key factors that limit the number of cores in CPU. Communication becomes increasingly more expensive in terms of time as the processor core increases. On the GPU side, the system does not automatically update the caches of other processing cores. It relies on programmers to write the output of each processor core to separate addresses. Actually, a single core is responsible for a single or small set of outputs. Moreover, adjacent memory locations are coalesced (combined) together by the hardware on GPUs, resulting in a single and more efficient memory fetch.

[10]Fermi and Kepler GPUs are equipped with a shared L2 cache, which is similar to the L3 cache function on the CPU.

2.2.5 Compute Unified Device Architecture (CUDA)

CUDA is an extension to the C language that allows GPU code to be written in regular C. The code is either targeted for the host processor (the CPU) or targeted at the device processor (the GPU). The host processor spawns multi-thread tasks (or kernels as they are known in CUDA) onto the GPU device. The GPU has its own internal scheduler that will then allocate the kernels to whatever GPU hardware is present.

CUDA enabled GPUs consist of a scalable array of multi-threaded streaming multiprocessors (SMs). Each SM contains 8 scalar processors (SPs), which can run simultaneously and executes identical instruction set. Up to 32 threads can be scheduled at a time, in a unit with a name "warp". There can be 24 warps active in one SM at most in the same time.

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory while the device refers to the GPU and its memory. Code running on the host manages memory on both the host and device, and also launches kernels, which are functions executed on the device. These kernels are executed by many GPU threads in parallel, which are organized in a grid-block-thread hierarchy. Threads within a block synchronize and cooperate with each other via fast block-wise shared memory. Threads from different blocks can only communicate through off-chip global memory with long latency. The grid is then formed by thread blocks that can be transparently deployed on various number of physical processors.

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA C program is:

- Declare and allocate host and device memory.
- Initialize host data.
- Transfer data from the host to the device.
- Execute one or more kernels.
- Transfer results from the device to the host.

In CUDA programming environment, a kernel function revoked by CPU is deployed to run on GPU. As displayed in Formula 2.3, a kernel call specifies the *execution configuration* using $\lll \ldots \ggg$ between the function name and the parenthesized argument list. Dg and Db define the thread dimensions for the grid and the blocks. Ns specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory and S relates to the associated stream.

$$kernalFunction \lll Dg, Db, Ns, S \ggg (para); \tag{2.3}$$

Moreover, CUDA has a hierarchy of memory space. Registers are thread-wise and on the top of this pyramid structure, which respond fastest within one processor cycle but are restricted by the limited number. Similarly, block-wise shared-memory are also on-chip and executes very fast. It is limited by the size as well. Constant memory and texture memory are off-chip but equipped with pretty fast caches. Lastly, accessing off-chip global and local memory cost several hundreds of cycles, though they are large in size.

Last but not least, CUDA programming model has been evolving with GPU architectures from Geforce, Tesla, Fermi to Kepler. And Maxwell will be released soon in 2013. The Tesla architecture is based on a scalable processor array. Several independent processing units called texture/processor clusters are employed to process the tasks. Fermi extends the performance and functionality of Tesla. Specifically, Fermi offers dramatically increased programmability and compute-efficiency through a series of architectural innovations. Recently, under the 28nm crafts, Kepler is the fastest and most efficient high performance computing architecture. It makes heterogeneous computing more accessible, with innovative SMX, dynamic parallelism and hyper-Q technology. The next generation GPU to Kepler will be the Maxwell, which has faster double precision speed and lower power consumption.

2.2.6 Alternatives to CUDA

Besides CUDA from NVIDIA, there are several alternatives in the GPGPU market. For example, OpenCL [5, 39, 17] is an open and royalty-free standard supported by NVIDIA, AMD, and other hardware manufacturers. The OpenCL trademark is owned by Apple, which sets out an open standard that allows the use of compute devices. CUDA is currently only officially executable on NVIDIA hardware while OpenCL supports all major brands of GPU devices, including CPUs with at least SSE3 support.

DirectCompute is Microsoft's alternative to CUDA and OpenCL. It is an application programming interface (API) that supports general purpose computing on GPUs on MS Windows 7 and Windows 8. DirectCompute is part of the Microsoft DirectX collection of APIs. The DirectCompute architecture shares a range of computational interfaces with its competitors, OpenCL and CUDA.

The main parallel processing language-extensions include MPI, OpenMP, windows threading model and pthreads. Firstly, as is mentioned in Section 2.2.1, MPI (Message Passing Interface) [20] is perhaps the most widely known messaging interface. MPI is a process-based parallel programming model. The parallelism is expressed by spawning hundreds of processes over a cluster of nodes and explicitly exchanging messages. Secondly, OpenMP (Open Multi-Processing) [12] is a system designed for parallelism within a computer system. The programmer specifies various parallel directives through compiler pragmas. The compiler then attempts to split the problem into N parts automatically, according to the number of available processor cores. OpenMP provides automatic scaling for the problems due to the underlying CPU architecture. The memory bandwidth in the CPU is the bottleneck for continuously streaming data. Thirdly, pthreads [38] is a library that is used significantly for multithread application. Using threads, pthreads is designed for parallelism within a single node. Moreover, the programmer should be responsible for thread management and synchronization, which provides more flexibility and consequently better performance for well-written programs. Fourthly, ZeroMQ(0MQ) [24] is a simple library designed for distributed computing that supports thread-, process-, and network-based communications models with a single cross-platform API. ZeroMQ provides dynamic connections and graceful fault-tolerant mechanism. Lastly, Hadoop [27] is an open-source version of Google's MapReduce framework [14]. In the map stage, Hadoop breaks (or map) a huge dataset into a number of chunks and split over hundreds or thousands of nodes using a parallel file system. Then in the reduce stage, the program is sent to the node that contains the data. The output is written to the local node. Subsequent MapReduce programs iteratively take the previous output and transform it in some way. Hadoop is a highly fault-tolerant and high-throughput system.

OpenACC is a set of "OpenMP-like" compiler directives for GPUs, which is supported by a number of compiler vendors [11]. With OpenACC, the programmer inserts a number of compiler directives marking regions as "to be executed on the GPU". The compiler then automatically moves data to/from the GPU and invokes kernels. Similar to the relationship between pthreads and OpenMP, CUDA provides the lower level of control and higher performance over OpenACC. Conversely, OpenACC requires a lower level of required programming knowledge, a lower risk of errors and shorter development time.

2.2.7 Parallelism with GPUs

A significant number of problems are known as "embarrassingly parallel", for which little or no effort is required to separate the problem into a number of parallel tasks. These types of problems can be implemented extremely well on GPUs and are easy to code. However, if one stage of the algorithm cannot be represented in this way, the

[11] PGI, CAPS, Cray, etc.

computation slows down due to the processors/threads spending more time sharing data than doing any useful work. The speedup will ultimately be limited. This stage turns out to be a bottleneck of this problem.

CUDA is ideal for an embarrassingly parallel problem, where little or no interthread or interblock communication is required. It supports interthread communication with explicit primitives using on-chip resources. Interblock communication is only supported by invoking multiple kernels in sequence, communicating between kernels using off-chip global memory.

CUDA splits problems into grids of blocks, each containing multiple threads. The blocks may run in any order and are allocated to any SM (symmetrical multiprocessors) that has free slots. If a grid of threads is analogous to an army of soldiers, the blocks are said to be like the units that are commanded by a lieutenant. The block is then split into several warps of threads, which is like a sergeant-lead squad of 32 soldiers. Figure 2.5 illustrates the CUDA-based hierarchy of threads view. The host program invokes the kernels to perform some action by providing some data. Each thread works on its individual part of the problem. Threads may communicate with each other by swapping data from time to time under the coordination of either the sergeant (the warp) or the lieutenant (the block). Any coordination with other blocks has to be performed by central command (the host or the kernel grid).

Thousands of threads orchestrate extremely high concurrency in this hierarchical manner. Actually, a typical modern GPU has on the order of 24K active threads. For example, a Fermi GPU has $65,535 \times 65,535 \times 1536$ threads in total, 24K of which are active at any time. To understand the parallelism of GPUs, several types of parallelism are defined as follows:

Definition 2.2.3. COARSE-GRAINED PARALLELISM IN GPU

Relative to fine-grained parallelism, bigger portions of processing element can be

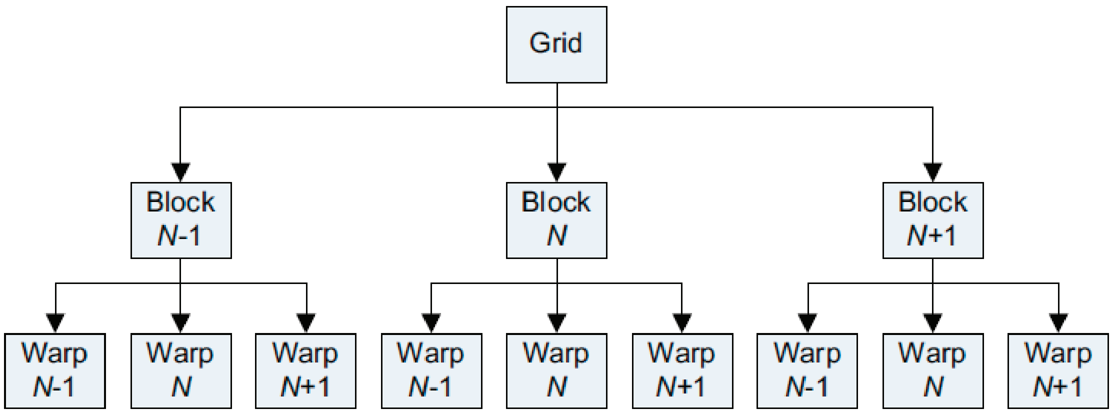


Figure 2.5: CUDA-based Thread View

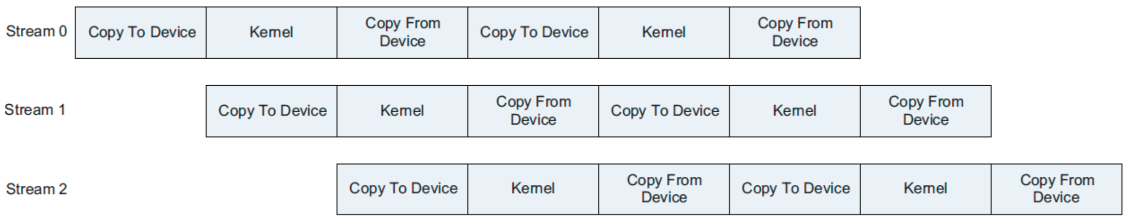


Figure 2.6: Stream Pipelining

employed to perform over a bulk of data.

GPUs support the coarse-grained parallelism pattern in two ways:

1. kernels can be pushed into a single stream and separate streams executed concurrently.

2. multiple GPUs can work together directly through either passing data via the host or passing data via messages directly to one another over the PCI-E bus.

As is defined in Definition 2.2.4, stream pipelining belongs to the coarse-grained parallelism on the GPUs. Figure 2.6 displays the partitioning of the tasks in GPU stream pipelining.

Definition 2.2.4. PIPELINE PARALLELISM

There are a number of powerful processors, each of which can perform a significant chunk of work. The output on one program provides the input for the next.

Besides coarse-grained parallelism, GPUs and CUDA can even support fine-grained parallelism which is defined in Definition 2.2.5. The CUDA parallel programming model has three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism (Definition 2.2.6). A problem is usually partitioned into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

Definition 2.2.5. FINE-GRAINED PARALLELISM IN GPU

Relative to coarse-grained parallelism, smaller portions of processing element can be employed to perform over fine-partitioning data.

Definition 2.2.6. TASK-BASED PARALLELISM

Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of program across multiple processors in parallel computing environments. Typically, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data.

Definition 2.2.7. DATA-BASED PARALLELISM

Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environments. Data parallelism focuses on distributing the data across different parallel computing nodes.

2.2.8 Parallel Patterns in CUDA Programs

There are several common parallel patterns in CUDA programs. Thinking in terms of patterns helps people to broadly deconstruct or abstract a problem. Therefore, learning and grasping well common parallel patterns enhance the efficiency of problem modeling and CUDA programming.

Loop-based patterns

A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop (the body of the loop) is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or indefinitely. Loops vary primarily in terms of entry and exit conditions (for, do...while, while), and whether they create dependencies between loop iterations or not.

Loop-based iteration is one of the easiest patterns to parallelize. With inter-loop dependencies removed, its then simply a matter of deciding how to split, or partition, the work between the available processors. This should be done with a view to minimizing communication between processors and maximizing the use of on-chip resources (registers and shared memory on a GPU; L1/L2/L3 cache on a CPU). Communication overhead typically scales badly and is often the bottleneck in poorly designed systems.

On the GPU the inner loop, provided it is small, is typically implemented by threads within a single block. As the loop iterations are grouped, adjacent threads usually access adjacent memory locations. This often allows people to exploit locality. Any outer loop(s) is(are) then implemented as blocks of the threads.

Fork/join pattern

The fork/join pattern is a common pattern in serial programming where there are synchronization points and only certain aspects of the program are parallel. The serial code runs and at some point hits a section where the work can be distributed to P processors in some manner. It then "forks" or spawns N threads/processes that perform the calculation in parallel. These then execute independently and finally converge or join once all the calculations are complete. This is typically the approach found in OpenMP and OpenACC, where a parallel region is defined with pragma statements. The code then splits into N threads and later converges to a single thread again.

The fork/join pattern is typically implemented with static partitioning of the data. That is, the serial code will launch N threads and divide the dataset equally between the N threads. The fork/join pattern is often used when there is an unknown amount of concurrency in a problem. Traversing a tree structure or a path exploration type algorithm may spawn (fork) additional threads when it encounters another node or path. When the path has been fully explored, these threads may then join back into the pool of threads or simply complete and wait to be re-spawned later.

GPUs have dynamic scheduling allocation. A block (thread) pool for GPUs is created for allocating tasks among SMs. Actually, this pattern is not natively supported on a GPU, as it uses a fixed number of blocks/threads at kernel launch time. Additional blocks cannot be launched by the kernel, only the host program. Thus, such algorithms on the GPU side are typically implemented as a series of GPU kernel launches, each of which needs to generate the next state. An alternative is to coordinate or signal the host and have it launch additional, concurrent kernels. Neither solution works particularly well, as GPUs are designed for a static amount of concurrency. Kepler introduces a concept, dynamic parallelism, which addresses this issue.

Tiling/grids

CUDA requires programmers to break the problem into smaller parts, each of which is then allocated to the processing elements present in the machine.

The tiling model is thus an easy model to conceptualize. Imagine the problem in two dimensions – a flat arrangement of data – and simply overlay a grid onto the problem space.

CUDA provides a simple two-dimensional grid model. For a significant number of problems, this is entirely sufficient. Considering a linear distribution of work within a single block, an ideal decomposition into CUDA blocks, however, is then demanded. As we can assign up to sixteen blocks per SM and we can have up to 16 SMs (30 on some GPUs), any number of blocks of 256 or larger is fine. In practice, we would like to limit the number of elements within the block to 128, 256, or 512, so this in itself may drive much larger numbers of blocks with a typical dataset.

Divide and Conquer

The divide-and-conquer pattern is also a pattern for breaking down large problems into smaller sections, each of which can be conquered. Taken together these individual computations allow a much larger problem to be solved. Typically divide-and-conquer algorithms are used with recursion. Most recursive algorithms can also be represented as an iterative model, which is usually somewhat easier to map onto the GPU as it fits better into the primary tile-based decomposition model of the GPU.

2.2.9 Hardware Overview

GPU hardware is radically different than CPU hardware. Notice the GPU hardware consists of a number of key blocks:

- Memory (global, constant, shared)
- Streaming multiprocessors (SMs)
- Streaming processors (SPs)

As is shown in Figure 2.7, a GPU device consists of one or more SMs. Virtually, GPU is really an array of SMs, each of which has N cores. This is the key aspect that allows scaling of the processor. The most significant part of SM is that there are multiple SPs within each SM. Each SM has access to a register file, which is much like a chunk of memory that runs at the same speed as the SP units. There is also a shared memory block accessible only to the individual SM that can be used as a program-managed cache. The shared memory is entirely under programmer control. Each SM has a separate bus into the texture memory, constant memory, and global memory spaces. Texture memory is a special view onto the global memory, which is useful for data where there is interpolation. Constant memory is used for read-only data and is cached on all hardware revisions. Like texture memory, constant memory is simply a view into the main global memory. Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card. Each SM also has two or more special-purpose units (SPUs), which perform special hardware instructions, such as the high-speed 24-bit sin/cosine/exponent operations.

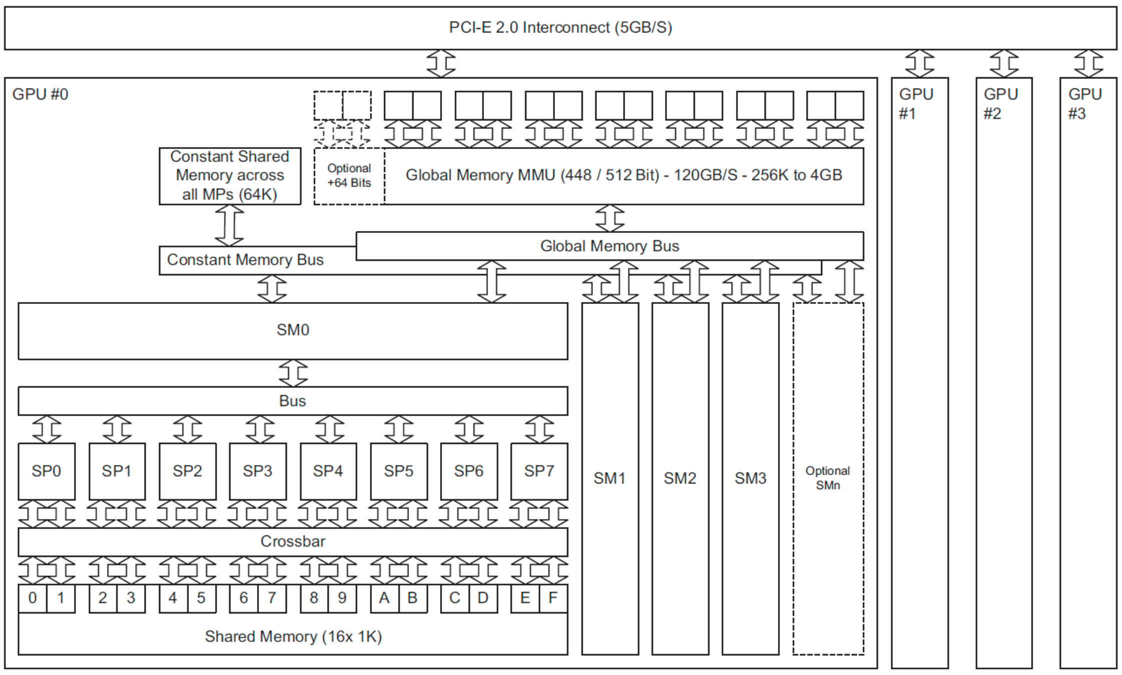


Figure 2.7: GPU Block Diagram

2.3 Related Work on Graph Processing on GPU

2.3.1 Graph Processing and Mining

Graph problems were first considered in the streaming environment by Henzinger et al. [23]. Then counting triangle number and estimating common neighborhoods are discussed in [8] and [10]. Both of these problems and those more "complicated" computation need to access the data in a very adaptive fashion. Since the entire graph is too large to be stored in the memory, emulating an aforementioned traditional algorithm necessitate a sequence of passes over the data. This has motivated data streaming models like the *Semi-Streaming* [18], *W-Stream* [15], and *Sort-Stream* [4] models. Actually, streaming is a useful method and an important computational model for handling large graphs that cannot be read into main memory. In streaming environment, data is normally accessed in a sequential fashion to bypass the memory limitation. Different from semi-streaming programming model [9], which is also named the external memory model [49], streaming algorithms abandon random access to the input graph data.

Given a graph data stream $S = \langle e_1, e_2, e_3, \ldots, e_n \rangle$, where $V = \{v_1, v_2, v_3, \ldots, v_m\}$, $E = \{e_1, e_2, e_3, \ldots, e_n\}$, and each item $e_i \in [m] \times [m]$, graph mining on streams is considered as estimating properties or finding patterns within the graph. According to [36], there are three common variants for graph mining on streams, namely multi-pass model, weighted/dynamic/directed graphs, and adjacency/incidence orderings.

Dense subgraph has significant more internal connections when comparing with its surrounding vertices. In the context of this thesis, the definition of a dense subgraph is the same with the one in paper [51], which is defined in Definition 2.5.2. Dense subgraph mining is close-relative but simpler when comparing with the traditional clustering which requires a strict partitioning of the graph [3]. There are three main types of dense graph mining algorithms, namely enumeration [32, 29], fast heuristic enumeration [19, 50] and bounded approximation [13, 7].

To enumerate all possible dense patterns is usually straightforward but time consuming. For some real applications, such as protein-protein interaction networks, fast heuristic methods are used to find the target pattern in large graphs. Moreover, bounding techniques are usually used to approximate the required density as well. In this sense, a series of famous algorithms were proposed, such as Singling Algorithm [19], GRASP algorithm [2], CSV algorithms [50], μ *-CompleteQB* [28], and some streaming-based algorithms [26].

Graph mining operators, such as computing graph statistics, graph matching, computing distance in a graph, graph random walk and graph triangulation, are basic graph mining approximative functional modules. These operators are fundamental to most of the graph mining applications, like PageRank and graph clustering. Graph triangulation operator walks through a graph and counts all triangles. The state-of-the-art triangulation operator is introduced in paper [9, 42, 51].

Paper [50] employed efficient bounding techniques on the mining of dense patterns such as clique, quasi-clique, and k-core components to approximate the indications of the dense patterns in the graph. In addition, a greedy heuristic algorithm is used to order all graph vertices into a linear fashion for graph traversal. Finally, a visual plot is provided to give clues about the size and distribution of the dense patterns. The time complexity of the algorithm CSV (cohesive subgraph mining and visualizing) is calculated as $O(|V|^2 log|V|2^d)$. For small and sparse graphs, the performance of CSV is polynomial, which is quite attractive when comparing with the existing algorithms.

2.3.2 Graph Processing on GPU

Modern GPUs have displayed an impressive computational power as well as higher memory bandwidth compared to CPUs. They are used by many graph (mining) applications as an accelerator to compute-intensive algorithms. Paper [21] presents a few fundamental algorithms - including breadth first search, single source shortest path, and all-pairs shortest path - using CUDA on large graphs. Paper [48] proposes an implementation of the push-relabel algorithm for graph cuts on the GPU. A minimum spanning tree algorithm on CUDA is presented in paper[47] as a recursive formulation of Boruvka's approach for undirected graphs. SimRank is a simple and influential measure of similarity between nodes in a graph. Paper [22] exploits the inherent parallelism and high memory bandwidth of GPU to accelerate the computation of SimRank on large graphs. Moreover, they use iterative aggregation techniques when computing SimRank scores concurrently for large graphs. Medusa [52] is a unified framework for supporting various graph computation and visualization operations on GPU.

2.3.3 Graph Processing Model

Many real life problems can be expressed in terms of connected entities. With graphical models, well defined graph theory can be utilized to process graph data and discover valuable results. There are three levels of general processing patterns for graphs, namely capture, query and mining. "Capture" identifies the relationship being generated or existing among entities. "Query" looks for some general information computed from the relationship among entities, which is just like a simple database query. "Mining" digs out some valuable knowledge from the information we can get using a series of data mining methodologies.

Since the graph size has been dramatically increased in the past years, how to store and process such a massive graph is a critical option. If the graph is kept in one large external storage, then the corresponding graph processing model works locally in shared memory system. Streaming algorithms need to be applied in this scenario. Otherwise, the graph would be broken down into multiple partitions and stored in different places.

Original algorithms are required to be redesigned in order to run in a distributed environment. Graph processing model can be sequential or parallel. A sequential processing model normally uses loops to iterate over the graph elements, while a parallel one launches processing concurrently among a set of graph elements. More and more graph processing models takes advantage of parallel programming techniques, which includes various multi-process, hyper-threading and many-threading methods.

Most parallel graph processing algorithms can be expressed in terms of a combination of "traversal" and "transformation". In graph "traversal" case, the algorithm walks through a path which consists of a sequence of segments. Marko and Peter's model [40] is an instance of such a case. The algorithm starts with walking from a set of vertices in parallel, and repeats until all segments are covered. While in graph "transformation" case, the algorithm can modify the graph by adding or removing vertices or edges. Pregel graph processing model is such an example. The algorithm also starts from some active vertices, executes some operations and repeats until all vertices become inactive. "traversal" and "transformation" are usually combined to express more complicated functions.

Parallel graph processing model can be synchronous or asynchronous. On one side, synchronization here means after some operations some parts of graph elements finish their tasks but waiting for other peers to finish. This timekeeping is for coordination purpose among all graph elements. A synchronization point is named as a barrier. Bulk Synchronous Parallel model [46] is a synchronous bridging model for designing parallel algorithms. All processors are connected by a communication network. After some local computation, the processor will send its results to other processors and wait for their messages to arrive. However, synchronization trades off the system performance with the model simplicity. On the other side, system can be implemented in an asynchronous model with careful data dependency design.

Graph processing model can be vertex centric or edge centric. Vertex centric means the model makes the vertices as its first class citizens and all computations are conducted by the vertices. In a parallel graph processing model, each thread or process represents a vertex for a vertex centric model. For an object-oriented programming language, a vertex can be modeled as a class object that is responsible for some computations and communications. An edge centric model takes the edges as its first class citizens, which is similar to a vertex model. An edge centric model can be converted to a vertex centric one since two connected vertices represent an edge. Google's Pregel [33] is a vertex centric model while Hama [43] makes edges as its first class citizens.

2.3.4 Graph Processing System

A graph processing system is an integrated system designed for graph processing and graph mining. A graph processing system has the functionalities of graph partitioning, graph traversal, graph merging and graph mining. There are several basic modules in a graph processing system that provides the fundamental functionalities, namely input/output module, partitioning module, combiner module, communication module, processing module and scheduler module. Boost Graph Library (BGL) [37] is a standard generic interface for traversing graphs and reusing basic graph algorithms and graph data structures. Parallel Boost Graph Library (PBGL) [16] is an extension to the BGL for parallel and distributed computing. PBGL offers distributed graphs and graph algorithms to exploit coarse-grained parallelism along with parallel algorithms that exploit fine-grained parallelism. Google's Pregel [33] is also a generic graph processing system for distributed computing environment. Several Pregel implementations have emerged in the literature recently. Phoebus [45] is an Erlang-based implementation of Pregel. Hama [44] is a distributed graph processing framework on Hadoop. Different from other pregel implementation, Hama is not a vertex centric model. GraphLab [30] is also a generic graph processing system that improves upon abstractions like MapReduce by compactly expressing asynchronous iterative algorithms with sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance.

2.4 Dense Neighborhood Graph Mining

This work introduces a new definition of dense subgraph pattern, the DN-graph. DN-graph considers both the size of the sub-structure and the minimum level of interactions between any pair of vertices. Detailed definitions and related work are attached in Appendix 2.5 at the end of this chapter. The mining of DN-graphs inherits the difficulty of finding clique, the fully-connected subgraphs. Thus, the DN-graphs can be approximately located using the state-of-the-art graph triangulation methods. The solution in this work consists of a family of algorithms, each of which targets a different problem setting. These algorithms are iterative, and utilize repeated scans through the triangles in the graph to locate the DN-graphs approximately. Each scan on the graph triangles improves the results.

Our iterative, triangulation-based approach has three advantages. First, most of the details involved in efficient processing, such as minimizing I/Os, are abstracted within the triangulation algorithm. The abstraction ensures this approach's extensibility to different input settings, e.g. when the target graph is too large to fit into memory, this approach only needs to change the access method of the graph links. In addition, the estimation of the local neighborhood is encapsulated within the triangulation algorithm. Second, as the estimation of the local density value improves with each additional iteration, users can adopt a "pay as you go" approach and obtain the most updated results on demand. Finally, when the graph is too large to fit into the main memory, statistics in the first iteration can be collected to support effective buffer management. There should be a need to store the local density value on a disk, since the triangles are generated in the same ordering in every iteration.

There is an algorithm family about triangulation based dense graph mining presented in this work. Their key features are shown in Table 2.1. For the ease of reference, we refer them as 1) TriDN, 2) BiTriDN and 3) StreamDN respectively.

	In Memory	Time	Space
TriDN	Yes	$O(klog\|V\|\|E\|^{\frac{3}{2}})$	$O(\|V\|log\|V\|+\|E\|)$
BiTriDN (Binary Bounding)	Yes	$O(klog\|V\|\|E\|^{\frac{3}{2}})$	$O(\|V\|log\|V\|+\|E\|)$
StreamDN (Semi-Stream)	No	$O(k\|E\|)$	$O(\|V\|)$

Table 2.1: A Family of DN-graph Mining Algorithms

There are variances between algorithms TriDN and BiTriDN that deal with in-memory graphs. The two algorithms vary from each other in the ways that iteratively refine λ to reach convergence. As briefly explained previously, the triangle based algorithm interactively uses the triangles to refine λ value. This process reaches convergence when all λ values remain the same as previous iteration's results.

The third algorithm, StreamDN, is for semi-streaming graph setting. To mine semi-streaming graphs, algorithm StreamDN applies the min-wise independent set property, which provides an approximation for triangulation using sequentially scan of graph edges, with bounded error.

Experimental data of our study in this work come from both theoretically proven data generators, as well as domain datasets. All the experiments are conducted on a workstation with a Quad-Core AMD Opteron(tm) processor 8356, 128GB RAM and 700GB hard disk. The operating system is Windows server 2003, Enterprize x64 edition.

In the experimental study, the efficiency (i.e., running time) of the DN-graph mining algorithms is evaluated. Figure 2.8 presents the effect of different graph density in a fixed parameter setting study. The synthetic graph generator G_{EC} varies the edge distribution by varying the embedded clique size c when $|V|$ (vertex size) is fixed. The trend over time

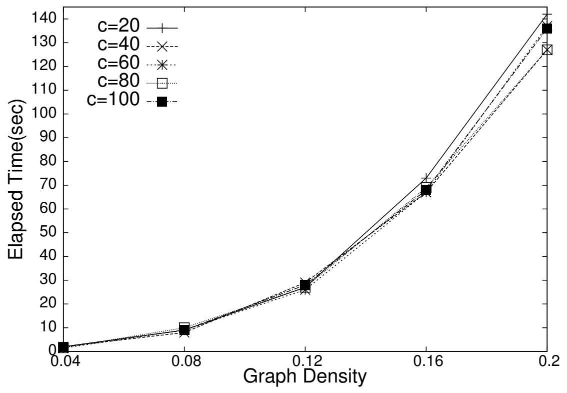


Figure 2.8: Vary graph density

roughly follows complexity $O(E^{\frac{3}{2}})$. This experiment has demonstrated that the program efficiency decreases exponentially as graph grows denser. When the graph arrives some degree of density, current system has reached its performance bottleneck. Methods with more computing power are potential solution for this situation. In the next chapter, we employ the state-of-the-art GPGPU techniques to provide additional computing power for graph triangulation acceleration.

2.5 Appendix

2.5.1 Preliminaries for DN-graph Mining

In the graph G, the neighborhood of a vertex v, is the set of vertices directly connecting to v. Formally, we denote it as $N(v) = \{u \mid (u,v) \in E\}$. The degree of vertex v can be given as $deg(v) = N(v)$. Moreover, the adjacency information for the graph is usually given as a set of vertex neighborhood: $adj(G) = \{N(u) \mid u \in V\}$. And the joint neighborhood is defined in 2.5.1.

Definition 2.5.1. JOINT NEIGHBORHOOD

The joint neighborhood of two vertices u and v, is the set of common neighbors of *the two vertices.*

$$N(u,v) = \{w \mid ((u,w) \in E) \wedge ((v,w) \in E)\} \tag{2.4}$$

The joint neighborhood of the subgraph G', is the set of all neighboring vertices not in G', but connecting directly to vertices within G'.

$$N(G') = \{u \mid (u,v) \in E_G \wedge (u,v) \notin E_{G'} \wedge u \notin V_{G'} \wedge v \in V_{G'}\} \tag{2.5}$$

Definition 2.5.2. DN-GRAPH

A DN-Graph with parameter λ, denoted $G'(V',E',\lambda)$, is a connected subgraph $G'(V',E')$ of graph $G(V,E)$ that satisfies the following conditions:

1. *Every connected pair of vertices in G' share at least λ common neighbors.*
2. *$\lambda(V' \cup \{v\}) < \lambda$, for any $v \in (V - V')$;*
3. *$\lambda(V' - \{v\}) \le \lambda$, for any $v \in V'$.*

Definition 2.5.3. DN-GRAPH SIZE: $\lambda(G')$

The DN-Graph size $\lambda(G')$ is defined as the number of vertices in the DN-Graph G'. According to the definition of the DN-Graph, the DN-Graph size minus two equals to the minimal joint neighborhood size (ν) between any two connected vertices within the graph.

$$\lambda(G') = \nu(G') + 2 = \min_{u,v \in V'} \nu(u,v) + 2 \tag{2.6}$$

As is stated in DEFINITION 2.5.2, a DN-Graph is a subgraph whose vertices share many common neighbors. The DN-Graph size (λ) relates to the minimal joint neighborhood size (ν) between any two connected vertices within the graph. The DN-Graph size for the subgraph G' is defined in DEFINITION 2.5.3. In addition, the DN-Graph size for an edge ($\lambda(e)$) and that for a vertex ($\lambda(v)$) are two local maximal density metrics for the subgraph. Obviously, the joint neighborhood size is equal to the number of the triangles the two connected vertices participates in, which is always an upper bound for the minimal joint neighborhood size. THEOREM 2.5.1 regulates the relationship between the DN-Graph size and the triangulation. $\widetilde{\lambda}(u,v)$ is defined to be an estimator for $\lambda(u,v)$.

Definition 2.5.4. SUPPORTING VERTICES: SP(E)

A vertex w is defined to be a supporting vertex for $\widetilde{\lambda}(u,v)$, iff:

$$\text{SP(E)} = \{\ w \mid \widetilde{\lambda}(u,v) \leq \min(\widetilde{\lambda}(u,w), \widetilde{\lambda}(v,w))\ \} \tag{2.7}$$

Definition 2.5.5. TRIANGULATION: $\Delta(G)$

The triangulation for a graph G is defined to be the technique of mining DN-G*raphs in the graph by computing the number of triangles (δ) each graph vertex and edge participate in.*

$$\Delta(G) = \{\ \lambda(e) \mid \forall e \in E\ \} \cup \{\ \lambda(v) \mid \forall v \in V\ \} \tag{2.8}$$

Theorem 2.5.1. TRIANGULATION FOR THE DN-GRAPH SIZE

According to the definition of the DN-G*raph, triangulation can be used to compute the* DN-G*raph size. The number of the triangles in which the edge (u,v) participates (a.k.a $\delta(u,v)$) is the upper-bound of the minimal joint neighborhood size $\nu(u,v)$. By tightening the upper-bound, the minimal joint neighborhood size can be reached when there are enough supporting vertices* SP(u,v) *(*DEFINITION 2.5.4*) for $\widetilde{\lambda}(u,v)$ in the subgraph G'.*

$$\lambda(u,v) = \nu(u,v) \leq \widetilde{\lambda}(u,v) \leq \delta(u,v) \tag{2.9}$$

2.5.2 DN-Graph As A Density Indicator

A graph $G(V,E)$ is a set of vertices V and a set of interactions E over $V \times V$. The size of graph G, denoted as $|V|$, is the number of vertices in V. The neighborhood of a graph vertex v, is the set of vertices directly connecting to v. $N(v)$ is used to represent it. If vertex u and v share some common neighbors, $N_\cap(u,v)$ represents the joint neighborhood. The neighborhood of e is the joint neighborhood of its two end vertices.

The joint neighborhood is denoted as $|N_e|$. For a subgraph G' of G, the neighborhood of G', $N(G')$, is the set of vertices $u \in G/G'$, which immediately connect with vertices in G'. Inside a graph, the measurement of minimal joint neighborhood size between any connected vertex pair is denoted as λ. The notation $\lambda(G)/\lambda(V)$ is used to refer to the measurement of a graph G with vertex set V. For brevity, the content inside the bracket is omitted and λ is used when the context is clear. $\tilde{a}$ is also used to represent an upperbound of quantity a. The upperbound of λ is thus written as $\tilde{\lambda}$.

In this work, a clique is a fully connected graph, in which every pair of vertices is connected by an edge. If the size of a clique is c, the clique is said to be a c-clique. When compared with clique of the same size, a quasi-clique has only a fraction (say δ) of edges in the graph, it is a δ quasi-clique. Conventionally δ is in the interval (0.5, 1].

Defined in Definition 2.5.2, a DN-graph should be a connected subgraph in which the lower bound of shared neighborhood between any connected vertices, λ, is locally maximized. Being a DN-graph, it has a local maximal λ value and the size of the DN-graph is maximized. This ensures that the DN-graph has more distinguishing power and maximal coverage. Similar with the graph's diameter and minimum cut, λ is an indicator of the graphs' underlying density. As proven in the appendix of Paper [51], it is a local maximum graph. For example, in figure 2.9, subgraph $ABCDEF$ is a DN-graph of λ value 3. If we include one more vertex A', the λ value of the graph $A'ABCDEF$ would drop significantly to 0. Similarly, taking away any vertex, say A, leads to a lower value λ. DN-graph is designed to represent dense patterns, as it captures subgraphs with more internal associations.

Besides the level of connectivity, a DN-graph also imposes restrictions on the minimal size of the shared neighborhood. This restriction is especially useful when predicting protein complexes via densely connected proteins within a protein-protein interaction (PPI) graph. A protein complexs formation often serves to activate or inhibit one or

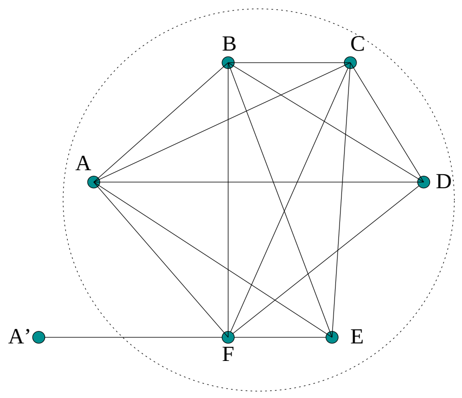


Figure 2.9: A DN-graph

more of the complex members. In a PPI network, we can observe the phenomenon that members of a protein complex share (significantly many) neighbors. The DN-graph definition reconciles the sharing of neighborhood.

Based on DN-graph, this work provides effective solutions towards mining DN-graphs within a massive graph. Generally speaking, the level of interactions among entities determine the density of the substructures. From this point of view, it is not surprising to see that some patterns are transformable to others. For example, a DN-graph is a more general case of a closed clique (Recall that a clique is a fully connected graph while the closed clique is the local maximal clique). In fact, a DN-graph is a relaxation of a clique, with less rigid size constraints. Lemma 2.5.1 states the relationship formally:

Lemma 2.5.1. *DN-graph and Closed Clique*
A graph contains a closed clique of size d if and only if the graph contains a DN-graph G with $\lambda = d - 2$ and $|G| = d$.

Using Lemma 2.5.1, people are able to reduce the close clique mining problem to DN-graph mining problem. The reduction signifies that DN-graph mining is NP-complete. Prompted by this result, we seek to develop heuristical solutions instead. Like the closed clique mining problem, the computational bottleneck for DN-graph mining is on counting degrees within a subgraph. In fact, the counting of local degrees relies heavily on the multiple joins of neighbors, which are computationally expensive. To avoid the complexity of multiple joins, we next introduce the concept of $\lambda(e)$.

As is discussed previously, the bottleneck of DN-graph mining is excessive number of the multiple joins of neighbors. This is because we have to test combinatorial number of subgraphs for their λ value and most subgraphs tested are not DN-graphs.

Definition 2.5.1. $\lambda(e)$
For all subgraph a graph edge e participates, the maximal λ value occurred is e's local density, We denote the maximal λ as $\lambda(e)$.

For example, in figure 2.9, let $e = (A, B)$, $\lambda(e)$ equals 3. Actually, all edges within the subgraph $ABCDEF$ have $\lambda(e) = 3$. The value $\lambda(e)$ indicates quantitatively, the most prominent relationships between two linked vertices. With the definition of local density, we next prove that using $\lambda(e)$ we are able to find all DN-graphs.

Theorem 2.5.2. *Locating DN-graph using $\lambda(e)$*
A graph G' is a DN-graph if and only if

- *all edges e within G' have equivalent λ value, λ_{max} and,*
- *for all $u \in N(G')$ and $v \in G'$, $\lambda(u, v) \leq \lambda_{max}$.*

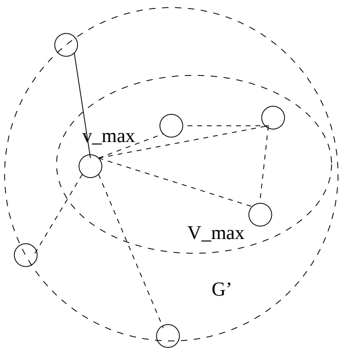


Figure 2.10: Proof of Theorem2.5.2

Proof. To prove the correctness of theorem 2.5.2, we use the abstract graph in figure 2.10. The complete proof consists of two steps. Firstly, G' must exist. Secondly, G' must contain some max-min DN graph. To prove the existence of G', we construct G' using graph vertices/edges and their λ values. First pick a vertex v with $\lambda(v) \geq \lambda(u)$ for all $(u \in N(v))$. Denote $\lambda(v)$ as λ_{max}. By the definition of local λ value, $\lambda(v)$ participates in a connected graph G' with $\lambda(G') = \lambda_{max}$. From v, we find all its immediately connected neighbors that have $\lambda(u) = \lambda_{max}$. From each u, we find u's immediately connected neighbors with local λ value λ_{max}. This process propagates until no such neighbor exists. The collection of discovered vertices form a connected subgraph G' with λ value λ_{max}.

Next, we show that G' contains a DN-graph. By first part of the proof, G' contains all vertices and edges with λ value λ_{max}. For a vertex $v' \in G'$, it only can form DN-graph of $\lambda = \lambda_{max}$ with vertices inside G'. If denoting the minimal set of vertices from G' that form an DN-graph with v' as V_{min}, the subgraph $V_{min} \cup v'$ is also a DN-graph. This proves that a graph G' containing the set of vertices with $\lambda(v) = \lambda_{max} > \lambda(u)$ where $u \in N(G')$ must participate in a DN-graph. The condition that $\lambda(v) = \lambda_{max}$ and $\lambda_{max} > \lambda(u)$, where u is the neighbor vertices of G', means the graph G' contains vertices with local maximal λ value. Since graph G' is always a super graph of some DN-graph, If a solution can find G', the DN graph can be located within G'.

With above two steps, we prove the correctness of theorem 2.5.2. $\square$

Based on Theorem 2.5.2, DN-graphs can be located by connecting edges with local maximal $\lambda(e)$.

Computing $\lambda(e)$ for all edges is however computationally prohibitive, as discussed in section 2.5.2. To facilitate approximation efficiently, we first find an upper bound value for $\lambda(e)$, the $\tilde{\lambda}(e)$, and then iteratively refine $\tilde{\lambda}(e)$ to capture the actual $\lambda(e)$ as accurately as possible.

The approximation is based on the fact that for an edge e, its $\lambda(e)$ value is upper bounded by the joint neighborhood size of the end vertices of e. This joint neighborhood size is in fact the number of triangles e participates in a graph. Thus we are inspired to use triangulation to approximate $\lambda(e)$ for every graph edge.

2.5.3 Triangulation Based DN-Graph Mining

A triangle consists of a vertex triple (u, v, w) and three edges (u, v), (v, w) and (u, w). The problem of counting or listing all triangles within a graph is referred as **Graph Triangulation** (Definition 2.5.5).

The joint neighborhood of edge $e(u, v)$ upper-bounds the local density $\lambda(e)$, while the number of triangles $e(u, v)$ participates in is equals to the joint neighborhood size. This indicates that graph triangulation provides an upper bound $\lambda(e)$ for every edge e. Here we use $\tilde{\lambda}(u, v)$ to represent the current upper bound of edge (u, v). What's more, given a graph triangle, the $\tilde{\lambda}(u, v)$ can tighten the other two edges' density upper bound. The following proposition gives the relationship between an edge e's ($\tilde{\lambda}(e)$) and its neighbors':

Proposition 2.5.1. *Neighbor Bounding of* $\tilde{\lambda}(e)$
Inside a triangle (u, v, w)*, if* $\tilde{\lambda}(u, v) \leq min(\tilde{\lambda}(u, w), \tilde{\lambda}(v, w))$ *we say* w *supports* $\tilde{\lambda}(u, v)$*.* $\tilde{\lambda}(u, v)$ *is valid if and only if:*

$$|\ \{w|w\ supports\ \tilde{\lambda}(u, v)\}\ | \geq \tilde{\lambda}(u, v)$$

The elementary operation behind local triangulation is the joining of vertex neighborhoods. The performance of a local triangulation algorithm heavily depends on the order of those join operations. In fact, it is a necessary preprocessing step to sort vertices according to their degrees for effective triangulation.

The triangulation algorithm generates triangles systematically for each edge of the graph. The generation of the triangles is a sequence of join operations between the neighbors of two connected vertices. Based on a special order of joining operations, the triangles are generated in a streaming fashion. The DN-graph mining algorithm thus obtains the local density information gradually along the triangle streams. Based on proposition 2.5.1, we can use the number of triangles an edge participates in $(TC(e))$ as the initial upper bound of the $\lambda(e)$, the $\tilde{\lambda}(e)$. To give an even more accurate bound for $\lambda(e)$, the algorithm uses the density value of e's neighbors' to validate the current upper bound $\lambda(e)$. Figure 2.11 shows how this process works graphically.

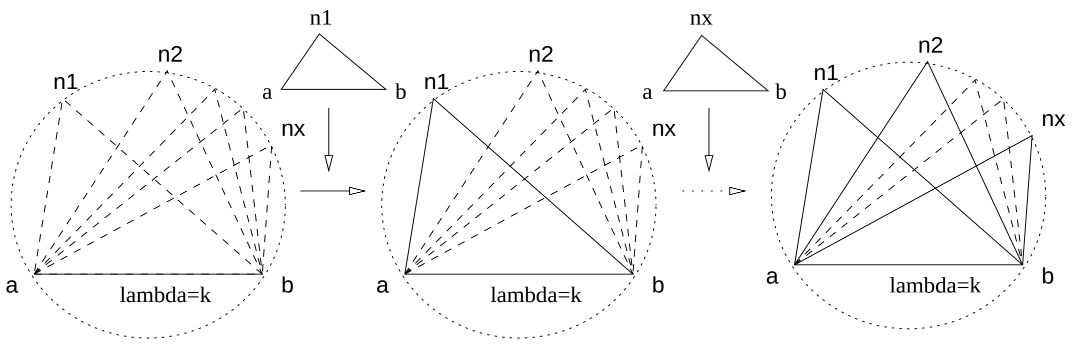


Figure 2.11: Use Triangle to Refine Local Density(λ)

In the first round of graph triangulation, we are aware of the triangular count of $e(a, b)$ (which is in fact $\tilde{\lambda}(e)$), and nothing about its neighbors. However, the triangular counts of the neighbors (a.k.a local density estimation) are available once the first round of graph triangulation is completed. To compute a more accurate $\tilde{\lambda}(e)$ for each edge, we will simply go through more rounds of triangulation and make use of the density information of the neighbors to further validate a new estimation of $\tilde{\lambda}(e)$ for each edge.

For a triangle $(a, b, n1)$, the algorithm checks whether the triangles $(a, b, n1)$ can possibly be a supporting evidence that edge $e(a, b)$ are in a DN-graph, with $\tilde{\lambda}(e)$. This is done by checking whether both the other two edges of triangle $(a, b, n1)$ (i.e. $e(a, n1)$ and $e(b, n1)$) have $\tilde{\lambda}$ greater or equal to $\tilde{\lambda}(e)$. If this is the case, this means that $n1$ is such a supporting vertex.

The triangle is then represented as a solid line indicating that $e(a, b)$ finds a new supporting vertex n1 in DN-graph with $\tilde{\lambda}(e)$. As new triangles approach, the algorithm counts the number of supporting vertices for edge (a, b) to form DN-graph, with the current value of $\tilde{\lambda}(e)$. After one pass of all triangles, the number of vertices that support each edge's density upper bound $\tilde{\lambda}(e)$ are available for further computation.

With the supporting neighbors' information, the algorithm is able to determine the upper bound of λ for each graph edge (the upper bound is denoted as $\tilde{\lambda}(e)$). If sufficient supporting vertices are found for $\tilde{\lambda}(e)$ for an edge $e(a,b)$, $\tilde{\lambda}(e)$ is a valid upper bound of $e(a,b)$'s λ value. If there is not enough supporting vertices for $e(a,b)$, the algorithm finds the next possible $\tilde{\lambda}(e)$ value and tests it in the next round of triangulation. The algorithmic description is given in Algorithm 1. Within the algorithm, $sc(e)$ records the number of vertices supporting current $\tilde{\lambda}(e)$ value.

Algorithm 1: Triangulation based DN-graph mining

```
   Input: Graph $G = (V, E)$
   Output: $\tilde{\lambda}(e)$ for each $e \in E$
1  Triangles = Triangulation($G$), $k(e)$ = Triangle_count($e$), iteration = 0 ;
2  while converge & $iteration \neq$ MAX_ITR do
3      sc = 0, converge = TRUE ;
4      forall the Triangles $(a, b, c) \in G$ do
5          if $e$ is supported then
6              $sc(e)$++ ;
7      forall the edges $e \in G$ do
8          if $sc(e) < \tilde{\lambda}(e)$ then
9              Find next possible value $\tilde{\lambda}(e)$ for $e$ ;
10             converge = FALSE ;
11     iteration++ ;
12 return $\tilde{\lambda}(e)$ ;
```

2.5.4 $\tilde{\lambda}(e)$ Bounding Choice

We can derive two variants of DN-graph mining algorithms from Algorithm 1, namely algorithms TriDN and BiTriDN. The two algorithms have different ways to decide the next possible $\tilde{\lambda}(e)$ value. The first variant, called TriDN, decreases $\tilde{\lambda}(e)$ by one (Line

9 in Algorithm 1 becomes $\tilde{\lambda}(e) = \tilde{\lambda}(e) - 1$), if current $\tilde{\lambda}(e)$ cannot obtain sufficient supporting vertices count. This strategy is useful when the triangle counts are close to the actual $\lambda(e)$ values (qualitatively, when $|TC(e) + 2 - \tilde{\lambda}(e)| \leq log\lambda(e)$).

When the triangulation results are far above the actual $\lambda(e)$ value, we can employ the second variant, called BiTriDN, which adopts a binary search strategy for the next possible value of $DN(e)$. BiTriDN requires additional information of possible $DN(e)$'s range. We use two numbers $lbk(e)$ and $\tilde{\lambda}(e)$ to record the lower bound and upper bound of $\lambda(e)$ value, and $mk(e)$ denotes the medium of range $[lbk(e), \tilde{\lambda}(e)]$. For completeness, we rewrite Line 7 onwards in Algorithm 1. BiTriDN has the advantage of fast convergence if the graph to be mined has many high degree vertices (qualitatively, when $|TC(e) + 2 - \tilde{\lambda}(e)| \geq log\lambda(e)$).

Algorithm 2: Binary DN-Graph Mining Variance "BiTriDN"

```
Input: Graph $G = (V, E)$
Output: $\tilde{\lambda}(e)$ for each $e \in E$
1 $mk(e) = k(e) = TC(e) + 2,\ lbk(e) = 2$ ;
2 Get support count $sc_{mk}(e)$ for all edges' $\tilde{\lambda}(e)$ ; // This part is the
      same as in Algorithm 1
3 forall the edge $e \in G$ do
4     if $sc_{mk}(e) < mk(e)$ & $lbk(e) < \tilde{\lambda}(e)$ then
5         $\tilde{\lambda}(e) = mk(e) - 1,\ converge = $ FALSE ;
6     else
7         $lbk(e) = mk(e)$ ;
8     $mk(e) = \frac{\tilde{\lambda}(e) + lbk(e)}{2}$ ;
9 return $\tilde{\lambda}(e)$ ;
```

2.5.5 Extension of DN-Graph Mining to Semi-Streaming Graph

The semi-streaming graph model assumes the vertices of the graph can be fitted into main memory, and the interactions among vertices are stored in an ordered manner within the secondary storage. While this assumption may not hold for arbitrarily large graphs, we can still handle up to Giga scale vertices (assume $|V|$ vertices require $|V|log|V|$ bits storage) with today's main memory capacities. Following the nature of physical storage devices, our streaming model assumes random access in primary storage (i.e. memory) and only sequential access in secondary storage. In the secondary storage, graph interactions are stored in the form of adjacency list. As a feasible solution towards a streaming graph $G(V, E)$, it should not exceed $log|V|$ scans of G's adjacency list.

In the semi-streaming graph setting, the exact triangulation algorithm cannot be directly applied in the DN-graph mining solutions. The information of the neighbors are stored in secondary storage and may not be immediately available when the algorithm retrieves it.

In view of above difficulty, our streaming solution first performs a semi-streaming triangulation, followed by the complete DN-graph mining solution in semi-streaming setting.

The neighborhoods join operations are in fact the process of determining the similarity between two sets. The most well-adapted measurement for set similarity is Jaccard coefficient. For two sets, A and B, Jaccard coefficient is calculated as $J(A, B) = \frac{|A \cap B|}{A \cup B}$.

In the semi-streaming graph setting, it is however expensive to calculate Jaccard coefficient between two neighborhoods since the operation of set joining requires expensive pre-processing of sets such as sorting or heap building.

In view of above difficulty, we use the property of min-wise independent set to approximate Jaccard coefficient. When dealing with large sets, min-wise independent property approximate set intersection size using sequential scan only.

Suppose A and B are defined on the set universe X, and π is a permutation over universe X, the min-wise independent property states: If $\pi[X]$ is a uniformly chosen random permutation over X, and $W \subset [X]$ is any subset over the universe, and $\pi[W]$ is the projection of W by permutation π, then the probability that two subsets' minimal projected images are equal is the same as the Jaccard coefficient. Formally, $P[min(\pi[A]) == min(\pi[B])] = J(A,B)$. Paper [9] proposes a streaming local triangle counting algorithm based on min-wise independent property.

The first step of algorithm StreamDN estimates local triangulation using edge scans. The next step is to calculate each edge's λ value using only edge scans 1. StreamDN, as presented in Algorithm 3, adopts the bounding process as algorithm BiTriDN. That is:

Algorithm 3: Streaming DN-Graph Mining Algorithm "StreamDN"

```
  Input: Graph $G = (V,E)$, $r$ : # of scans of graph links, $k$ : # of bits for hash
         values
  Output: $\tilde{\lambda}(e)$ for each $e \in E$
1 $mk(e) = \tilde{\lambda}(e) = TC(e),\ lbk(e) = 0$ ;
2 Triangulation and store triangle count $TC(v,u)$ for all $e \in E$ as in algorithm 5 in
  appendix. while $!converge \ \&\ iteration \neq$ MAX_ITR do
3     $sc_k = 0,\ ubk(e) = \tilde{\lambda}(e) = TC(e),\ lbk(e) = 0$ ;
4     forall the edge $(u,v) \in G$ do
5         $sc_k(u,v)$ = number of $u$'s neighbor with $\tilde{\lambda}(u,v)$ ;
6         Bound $\tilde{\lambda}(u,v)$ using $ubk(u,v)/lbk(u,v)/sc_k(u,v)$ ;
          // the same as Algorithm 2
7 return $\tilde{\lambda}(e)$ ;
```

The only difference between the streaming version of the algorithm and BiTriDN is when counting the supporting vertices. In StreamDN, we can only access the graph edges sequentially. In view of the restriction, proposition 2.5.1 is relaxed to as follows:

Proposition 2.5.2. ***Relaxed*** *Neighbor Bounding of* $\lambda(e)$

Given a graph edge $e(u,v)$ and the joint neighbor set $N_\cap(u,v)$, we say a vertex $w \in N_\cap(u,v)$ is a supporting vertex of $\tilde{\lambda}(e)$ ***if*** *$\lambda(u,w) \geq \tilde{\lambda}(e)$. An integer k is a valid upper bound of $\tilde{\lambda}(e)$ if and only if there are at least k of such supporting vertices in $N_\cap(u,v)$.*

Chapter 3

Streaming and GPU-Accelerated Graph Triangulation

In this chapter, we take advantage of the state-of-the-art GPGPU technology to accelerate and scale iterative triangulation. Facing the challenges of applying efficient triangulation over a massive graph, we propose a parallel triangulation algorithm across heterogeneous platform and achieve a notable speedup. We first utilize a streaming partition to divide a massive graph. To facilitate inter-partition communications, we then design a message spreading mechanism. Moreover, we employ a streaming pipelining strategy to speedup data transfer among different memory spaces. Last but not least, several novel GPU graph data structures are designed to enhance the graph processing efficiency on the GPU.

3.1 Problem Statement

Triangulation is defined to be the technique of mining dense sub-components in the graph by computing the number of triangles each graph vertex and edge involve. However, exact counting of triangles in large graphs is computationally expensive. Since graphs are growing larger and larger, graph triangulation has become a huge burden for general computer systems. Limited memory space and computing power have become the main bottlenecks of the system. We named this situation as hitting “memory wall” and “power wall”.

In order to overcome the memory limitation, researchers proposed several solutions that extend the algorithm operating space. One method is to apply the semi-streaming model that stores the data in the external memory and saves the vertex list in the memory. However, semi-streaming model only converts the problem from “memory-bound” to “IO-bound”. The other way is to adopt the divide and conquer strategy which partitions the graph into sub-portions, each of which is then read into the memory for triangulation. Nevertheless partitioning is a feasible approach, additional efforts should be exerted to process the inevitable redundant vertices and edges that originally link two sub-portions. Both of the aforementioned answers to the “memory wall” problem seem to be less efficient. In consideration of the multi-core CPU architecture, we combine the streaming model and the pipelining approach to speedup the data transfer.

In order to climb over the “power wall”, one solution is to parallelize the triangulation algorithm for multi-core CPU systems. SIMD programming model can be applied over the algorithm design. Multi-threading and processor affinity techniques are adopted to increase the virtual and real parallelism of triangulation. However, multi-core CPU was still unable to provide enough real parallelism for triangulation over increasing massive graphs. There are only 4 to 16 cores in a recent personal computer systems. No matter how many threads we can create for a large graph triangulation, the actual parallelism is constrained by the physical parallelism provided by the CPU. In order to exploit the potential computing power of general computer systems, we naturally turn to heterogeneous computing. In this chapter, we take advantage of the state-of-the-art GPGPU techniques to accelerate and scale the iterative triangulation.

The rest of this chapter is organized as follows: Iterative triangulation is introduced in Section 3.2 and a parallel version of triangulation is proposed in Section 3.3. The message spreading mechanism is designed in Section 3.4 to assist inter-subgraph communications. We then describe a streaming partition strategy in Section 3.5. Moreover, Section 3.6 proposes an efficient streaming pipelining approach. Dynamic threading techniques are utilized in Section 3.7. Several novel graph data structures designed for GPU are discussed in Section 3.8. Section 3.9 proves the correctness of parallel triangulation. Finally, experiments are displayed in Section 3.10 before a conclusion is made.

3.2 Iterative Triangulation

As stated in THEOREM 2.5.1, triangulation can be used to compute the DN-Graph size. Initially, the triangle count δ for each edge e is computed by calculating the joint neighborhood size of the two ends of the edge. In addition, according to THEOREM 2.5.1, $\delta(e)$ is a tight upper-bound for the minimal joint neighborhood size $\nu(e)$. Specifically, the estimation of the minimal joint neighborhood size, $\widetilde{\lambda}(e)$, can be bounded by the current $\widetilde{\lambda}$ values of the edge e's two neighboring edges, according to the definition of the supporting vertex (DEFINITION 2.5.4). Similarly, $\widetilde{\lambda}(e)$ is also used to bound the $\widetilde{\lambda}$ value of e's neighbors. When a graph flows through the processors, all edges are affected by their neighbors and also influence their neighbors in reverse. During an iteration, one edge e is only bounded once. Those neighbors of this edge whose $\widetilde{\lambda}$ values are updated later can only affect e with their new $\widetilde{\lambda}$ in the next iteration. By iteratively streaming through the graph, $\widetilde{\lambda}(e)$ approaches a steady state, in which e together with some of its neighbors forms a DN-Graph.

EXAMPLE 3.1 is a simple working instance to illustrate iterative triangulation.

Example 1. *Figure 3.1 displays a graph G with 7 vertices. By counting the number of triangles in which each edge participate, the $\widetilde{\lambda}$ values of all edges are initial-*

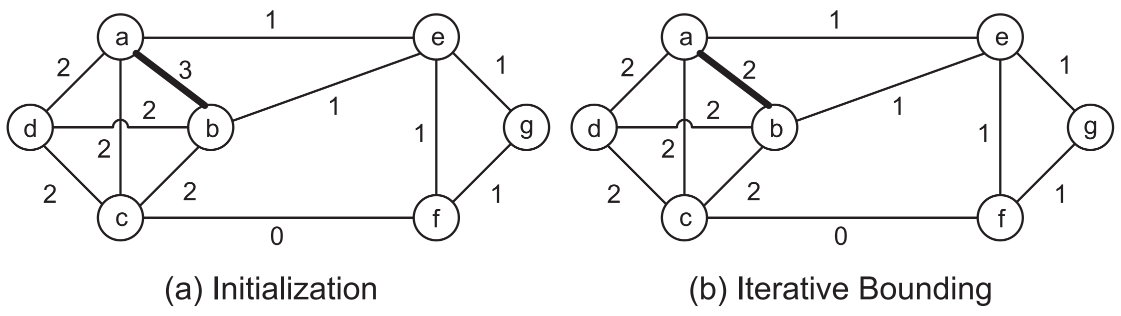


Figure 3.1: Iterative Triangulation

ized to their δ values. This is shown in figure 3.1(a). The edge (a,b) is involved in three triangles $\{\triangle_{abc},\triangle_{abd},\triangle_{abe}\}$, so $\widetilde{\lambda}(a,b)$ is initialized to be $\delta(a,b)$=3. Since $\widetilde{\lambda}(a,b) >$ $\max(\widetilde{\lambda}(a,d), \widetilde{\lambda}(b,d), \widetilde{\lambda}(a,c), \widetilde{\lambda}(b,c), \widetilde{\lambda}(a,e),\widetilde{\lambda}(b,e))$, there is no supporting vertex for the edge (a,b). $\widetilde{\lambda}(a,b)$ is then reduced to 2 as displayed in Figure 3.1(b). In the second iteration, the $\widetilde{\lambda}$ values of all edges in the subgraph $G'\{a,b,c,d\}$ have enough supporting vertices, falling into a steady state. Hence, $\lambda(a,b)$=$\widetilde{\lambda}(a,b)$=2 and the subgraph G' forms a DN-G*raph before the algorithm halts.*

In order to compute the λ value for each edge in a graph, triangulation algorithm examines all edges and their neighbors once an iteration. In other words, there may be $O(|E|)$ join operations incurring intensive computation for each iteration. In practice, we notice that when the graph to be processed becomes larger and larger the performance of the system drops exponentially. Lacking effective computing power has become the main factor that limit the scalability of traditional triangulation algorithm. Additionally, programs always crash as input graphs are too large to be held in the memory. A semi-streaming solution might prevent programs from crashing, such as stated in paper [51], by saving only the vertices in the memory. Nevertheless, the limited data transfer rate of the system I/O becomes the system bottleneck. Processors always stall their operations while waiting for graph data to be loaded or unloaded from the external storage.

In the next section, we will first try to explore additional computing power by exploiting the potential parallelism out of the graph triangulation. After that, we will introduce a combination of streaming and pipelining techniques to solve the memory-bound problem.

3.3 Parallel Triangulation

Iterative triangulation was introduced in paper [51] as a “pay as you go” approach to locate the dense subgraph pattern approximately. As a sequential algorithm designed for small graphs, iterative triangulation faces scalability problem when increasing graph size exhausts the existing computing power. In this section, we propose a parallel version of triangulation to efficiently mine dense subgraph patterns. Heterogeneous computing and the state-of-the-art GPGPU techniques are employed to parallelize the algorithm.

One step back, to explore the performance of the application, let’s reflect the problem again and analyze the parallelism of the application. As mentioned previously, the λ values for a graph G are computed by triangulation. An initial value δ for each edge is calculated by counting triangles. Based on this upper-bound, an iterative bounding process is applied on the estimated value $\widetilde{\lambda}$ to approach the λ value for all elements in the graph. For a massive graph that cannot directly resides in the memory, an efficient partition approach divides the graph into subgraphs. Triangulation should be conducted concurrently on these subgraphs since a traversal across the graph may start from any subgraph. Furthermore, triangulation can also be applied on different vertices/edges simultaneously. With a neighboring list, each vertex has little inherent data dependency to each other. All operations executed by the vertex are said to be data parallel. We will delve into the aforementioned two levels for parallelism when parallelizing triangulation.

As introduced in SECTION 2.2.5, CUDA is a parallel programming model for general purpose computation on GPUs. Being a many-core co-processor to CPU, NVIDIA GPUs are optimized hardware for data parallel operations. A batch of light-weight threads can be launched to run concurrently on GPU parallel hardware at the same time. People always send the most compute-intensive part of algorithms to be processed on GPU. The number of computing cores and hybrid system architecture decide the real (hardware) parallelism. The efficiency and performance of the application are maximized when algorithm (virtual) parallelism adapts to the hardware parallelism.

To parallelize triangulation, a massive graph is initially divided into several subgraphs. Then a number of threads take charge of these partitions, each of which is then scheduled to run on one physical processor. A join operation that computes the common neighbors for the two ends of an edge is employed for triangle counting. When a graph stream flows through processors, the join operations are executed in parallel. CPU threads as well as GPU threads can be assigned for this task. For example, we have tried to use one CPU thread to handle one partition and launched a pile of GPU threads for vertices to join their neighbors. Several combinations of CPU threads and GPU threads can be designed according to the algorithm and system architecture. Similarly, we parallelize iterative bounding using simultaneous CPU or GPU threads.

ALGORITHM 4 is the parallel version of iterative triangulation. The algorithm reads in a stream of the graph and output the λ values for all graph elements. In the first line, the graph stream is divided into an array of sub-streams. Line 3 ~ 6 initialize the $\widetilde{\lambda}$ value for each graph element using its δ value. Iterative bounding is enclosed with the "**repeat...until**" loop starting from Line 7 onwards. By adopting "pay as you go" approach, the iteration continues until all bounding operations converge in every partition or the maximum number of iterations has been reached. As is analyzed above, different threads can drive the iterative bounding concurrently. $cvg[i]$ denotes whether the bounding has ceased in the ith partition $P[i]$. Line 11 swaps in the adjacency and the $\widetilde{\lambda}$ information from the external disk. If any message has been received from $N(P[i])$,

Algorithm 4: Parallel Triangulation

```
Input: $G = (V, E)$
Output: $\lambda(G)$
$P(V', E')$ = Graph_Partition($G$) ;
int $itr\_count$ = 0, $\widetilde{\lambda} = 0$ ;
foreach graph partition $P[i](V'[i], E'[i])$ do
    foreach Edge $e(u, v) \in E'[i]$ do
        $\widetilde{\lambda}(e) = \delta(e) = |N(u, v)| = Join(N(u), N(v))$ ;
        $\widetilde{\lambda}(u) = \max(\widetilde{\lambda}(u), \widetilde{\lambda}(e))$, $\widetilde{\lambda}(v) = \max(\widetilde{\lambda}(v), \widetilde{\lambda}(e))$;
repeat
    bool $converge$ = TRUE ;
    foreach graph partition $P[i](V'[i], E'[i])$ do
        if $cvg[i]$ = FALSE then
            Swap in $adj(V'[i])$, $\widetilde{\lambda}(P[i])$ from disk ;
            if $(i \neq 0) \parallel (itr\_count \neq 0)$ then
                Swap in $msg[i]$; Update($adj(V'[i])$, $\widetilde{\lambda}(P[i])$, $msg[i]$) ;
            dim3 $Db$ = BLOCKSIZE, $Dg = \frac{V'[i]}{Db}$ ;
            ItrBound<<<Dg,Db>>>($\widetilde{\lambda}(P[i]), msg[i], cvg[i]$)
                int idx = blockIdx.x × blockDim.x + threadIdx.x ;
                int support_cnt = 0 ;
                __shared__ bool SCVG$[Db]$ ;
                __shared__ list< pair<int,int> > SMSG$[Db]$ ;
                foreach vertex $v \in N(idx)$ do
                    while $u \in N(index, v)$ do
                        if $\widetilde{\lambda}(v, idx) \leq \widetilde{\lambda}(u, v)$&&$\widetilde{\lambda}(v, idx) \leq \widetilde{\lambda}(u, idx)$ then
                            support_cnt++ ;
                foreach vertex $v \in N(idx)$ do
                    while $u \in N(index, v)$ do
                        if $\widetilde{\lambda}(v, idx) \leq \widetilde{\lambda}(u, v)$&&$\widetilde{\lambda}(v, idx) \leq \widetilde{\lambda}(u, idx)$ then
                            support_cnt++ ;
                    while $u \in N(index, v)$ do
                        if support_cnt(v,idx)< $\widetilde{\lambda}(v, idx)$ then
                            $\widetilde{\lambda}(v, idx)--$ ;
                            if $v \in N(P[i])$ then
                                Generate(SMSG, $v, \widetilde{\lambda}(v, idx)$) ;
                            SCVG$[threadIdx]$ = FALSE ;
                __synchronized();
                $cvg[i] = Rd(Rd($SCVG,threadIdx.x),blockIdx.x$)$ ;
                $msg[i]$ = Encapsulate(SMSG) ;
            Message_Send($msg[i]$) ; Swap out $\widetilde{\lambda}(P[i])$ to disk ;
        $converge$ &= $cvg[i$++$]$ ;
    $itr\_count$++ ;
until converge = TRUE && itr_count > MAX_ITR;
```

the $\widetilde{\lambda}$ value of some graph element will be updated in Line 14. Noticeably, the routine `ItrBound` is the kernel function that is deployed to run on GPU. After iterative bounding, messages are sent to $N(P[i])$ and $\widetilde{\lambda}(P[i])$ are swapped out so that the resident data in the memory can be minimized.

The kernel function `ItrBound` is invoked with the execution configuration (enclosed by triple angle brackets "⋘⋙") and some algorithmic parameters. Specifically, the dimension of thread blocks Db is preset to be *BlockSize* and the dimension of thread grids Dg is calculated as the subgraph vertex size divided by Db. Thus, the GPU threads are put into one-to-one correspondence with the vertex set of the subgraph. With the global thread index, the vertex is connected by the GPU thread. It is denoted as a thread vertex. In addition, those algorithmic parameters in parentheses are declared by cpu but allocated on gpu memory, which link the data across hybrid platforms. The first "*foreach*" loop counts the supporting vertices for all edges formed by the thread vertex and its neighbors. The second "*foreach*" loop decreases the $\widetilde{\lambda}$ value for each aforementioned edge if there is no enough supporting vertices. If a neighbor v connecting to this thread vertex idx happens to be a peripheral boundary vertex, which means $v \in N(P[i])$, a message is generated for the edge (idx, v). A synchronizing barrier is set in Line 32 to wait for all threads to complete their bounding tasks. Before exiting the device code, the converging flags for all thread are reduced to the global value $cvg[i]$. And the messages generated by each thread are collected into the messaging list $msg[i]$.

Different from traditional sequential algorithms, there are several types of memory spaces utilized in the ALGORITHM 4. Firstly, massive graphs (G,$P[i]$) and related data ($\widetilde{\lambda}$, $adj(V'[i])$, etc.) are stored in the external disk. However, direct access to the disk always results in excessive IO costs. We employ the disk as a supplementary storage. Secondly, as a major place for data processing and scheduling, main memory can be accessed by multi-core CPU for partitioning, initial triangle counting and message spreading. Besides host memory, GPU global memory is the device memory connecting to main memory via a high speed bus[1]. Similar to main memory, GPU global memory acts as media for data processing and scheduling on the device. msg[i], cvg[i] and intermediate data are saved in global memory while the kernel function is executed. In addition, texture memory is the auxiliary device memory that can be accessed via fast cache by GPU processors. $\widetilde{\lambda}(P[i])$ is read-only during the execution of the device code. We combine $\widetilde{\lambda}(P[i])$ with the texture memory to speed up parallel data access on the device. Furthermore, constant memory is another read-only device memory that we assign our constants (BLOCKSIZE, etc.) there. Last but not least, shared-memory is the block-wise fast device memory. Different from other memory types, shared-memory is on-chip and close to GPU processors. Restricted by the limited size, we only employ two shared arrays in the shared memory for fast data updates. The shared array SCVG is used for flagging whether any $\widetilde{\lambda}$ value of the edges incident to the thread vertex has been updated. The reduce function can be conducted from the block-wise and grid-wise levels to combine the flagging values and save the result in a global variable. The shared array SMSG records the updated $\widetilde{\lambda}$ values and the corresponding edge information for the thread vertex.

The parallel triangulation algorithm streams the graph from the disk to GPU memory, which takes $O(|E|)$ time complexity. Preprocessing sorts vertex and adjacency list into descending order of degrees, which costs $O(|V|log|V|)$ time complexity. After that, it counts triangles within the graphs for each vertex concurrently. For a vertex, its neighborhood size is at most constant (say S). The counting over all edges require $O(S|E|)$ time. Assume setting $S = \sqrt{|E|}$, taking into consideration of a fixed number of iteration k, the time complexity for parallel triangulation is $O(k\frac{|E|^{\frac{3}{2}}}{|V|})$. If insisting on convergence, the algorithm may need up to $O(k|E|^{\frac{3}{2}})$. As we apply binary search paradigm to test pos-

[1]PCI-Express bus: 4GB/s for v1.x

sible $\lambda(e)$ for every e, we adjust the time complexity to $O(k\frac{log|V|\times|E|^{\frac{3}{2}}}{|V|}) = O(k\frac{|E|^{\frac{3}{2}}}{|V|})$.

As is introduced above, parallel triangulation bases on partitioning and iterative bounding. The communication among neighboring partitions and between successive iterations is important for the effectiveness and efficiency of the algorithm. The next section is a specific discussion about the message spreading mechanism in parallel triangulation.

3.4 Message Spreading Mechanism

Parallel triangulation partitions a massive graph and explores the iterative bounding concurrently. As is introduced in the last section, any operation exerted on the graph elements have influence on the neighbors of these elements. According to THEOREM 2.5.1, the $\widetilde{\lambda}$ value of an edge or a vertex is utilized to bound the corresponding values of its neighboring edges or vertices. For example, in one iteration, the $\widetilde{\lambda}$ of an edge e is employed to bound all its neighboring edges. An update to the $\widetilde{\lambda}$ value of its neighbors will reversely affect $\widetilde{\lambda}(e)$ in the next iteration. If some of these neighbors happen to be in a different partition, an efficient message spreading mechanism is needed to exchange the update information of $\widetilde{\lambda}$ values among subgraphs.

The message spreading mechanism in our parallel triangulation algorithm consists of four intermediate components, namely GENERATE, ENCAPSULATE, MESSAGE_SEND and UPDATE. They are located respectively at Lines 14, 30, 34 and 35 of ALGORITHM 4. ALGORITHM 5 lists how these four routines function. The `Generate` function is used to generate a message whenever the $\widetilde{\lambda}$ value of the cut-edge e has been updated. A piece of message msg is declared to be an integer pair: (v_{dest}, $\widetilde{\lambda}(e)$). The SMSG array is a list of integer pairs defined in GPU shared memory for fast recording the update information. Using *threadIdx* to index threads within block-wise shared memory, the message msg is inserted into the SMSG array. The `Encapsulate` function attaches the source information and the destination partition to the message. Then all messages are transferred by GPU threads to the $msg[i]$ array in the global memory. After all messages are inserted into the $msg[i]$ array, a sort function is invoked on the partition number within the SMSG array. This shuffling function groups those messages with the same destination partition number. This mechanism reduces the IO cost while accessing message files from external disk on general computer systems. The `Message_Send` function then sends the groups of messages to the corresponding message file $msg[pid]$ of the destination partitions. Finally, the `Update` function reads in the message file that contains the messages received by the current partition from its neighboring partitions. Then each piece of the message is analyzed before all $\widetilde{\lambda}$ values of the corresponding edges and vertices within the subgraph are updated.

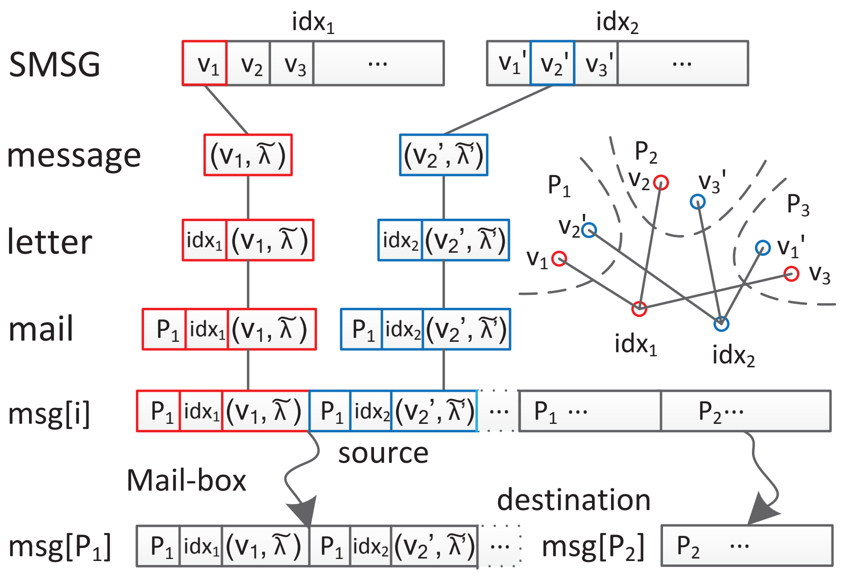


Figure 3.2: Message Spreading Mechanism

To design an efficient communication mechanism, we model message spreading as the process of mail delivery. As an illustrating example, FIGURE 3.2 displays how the messages are delivered across partitions. Thread vertices idx_1 and idx_2 both have edges connecting to the vertices in other partitions. When $\widetilde{\lambda}(idx_1, v_1)$ and $\widetilde{\lambda}(idx_2, v_2')$ are updated, messages <v_1,$\widetilde{\lambda}$> and <v_2',$\widetilde{\lambda}'$> are generated within SMSG shared array by `Generate`. Then messages are formed as a "letter", when the source vertex is signed as a signature. Moreover, the "letter" is further encapsulated as a "mail", with the destination partition number attached. All of these "mails" are then transferred by GPU threads into the "mail-box" $msg[i]$ residing in the global memory. Before sending, all mails in this "mail-box" are shuffled according to their destination partition numbers. Since the destination vertices v_1 and v_2' belong to the same partition P_1, their mails are grouped in the same "package". Finally, the "package" containing the two messages is then sent back to the destination "mail-box" $msg[P_1]$, which has been swapped into the main memory.

3.5 Large Graph Partitioning

The *graph partitioning problem* usually divides a large graph $G(V, E)$ into several subgraphs $P(V', E')$ with a smaller size. The set of graph vertices and edges are then partitioned into smaller components with specific properties. Graph partitioning greatly increases the parallelism of an application, which makes it more and more important for large scale and distributed applications. An objective function is usually defined to quantify the quality of the partition algorithm. For example, a good partition may be defined as one in which the number of edges connecting separated components is small. Unfortunately, graph partitioning has been proven to be an NP-hard problem. All well-known practical partition algorithms give sub-optimal approximative solutions. In our context of parallel triangulation on personal computers, massive graphs are always too large to reside in the memory. We employ efficient large graph partitioning algorithms to divide and conquer the problem. Instead of using semi-streaming partition algorithms,

Algorithm 5: Message Spreading

```
1  Generate (SMSG, $v$, $\widetilde{\lambda}(v, idx)$)
2      pair<int,int> msg= pair($v$, $\widetilde{\lambda}(v, idx)$) ;
3      SMSG$[threadIdx]$.insert(msg) ;
4  Encapsulate (SMSG)
5      forall the message ms ∈ SMSG$[threadIdx]$ do
6          pair<int,pair<int,int>> letter = pair(idx, ms) ;
7          pid = Index(SMSG$[threadIdx]$.first) ;
8          pair<int,pair<int,pair<int,int>>> mail = pair(pid, letter) ;
9          $msg[i]$.insert(mail) ;
10     sort($msg[i]$, first);
11 Message_Send ($msg[i]$)
12     forall the mail ml ∈ $msg[i]$ do
13         if $pid \neq ml.first$ then
14             $pid = ml.first$ ;
15             Swap in $msg[pid]$ from disk ;
16         $msg[pid]$.insert($ml$) ;
17 Update ($adj(V'[i])$, $msg[i]$)
18     forall the mail ml ∈ $msg[i]$ do
19         $\widetilde{\lambda}(ml)$ = ml.second.second.second ;
20         u = ml.second.first, v = ml.second.second.first ;
21         e = (ml.second.first, ml.second.second.first) ;
22         if $\widetilde{\lambda}(ml) < \widetilde{\lambda}(e)$ then
23             $\widetilde{\lambda}(e) = \widetilde{\lambda}(ml)$ ;
24         if $\widetilde{\lambda}(ml) < \widetilde{\lambda}(u)$ then
25             $\widetilde{\lambda}(u) = \widetilde{\lambda}(ml)$ ;
26         if $\widetilde{\lambda}(ml) < \widetilde{\lambda}(v)$ then
27             $\widetilde{\lambda}(v) = \widetilde{\lambda}(ml)$ ;
```

we take advantage of streaming methods to process a large graph. Four disk-based graph partitioning algorithms are proposed and compared for efficiency as follows:

Vertex-centric Partition assign all vertices to each subgraph uniformly, according to their indices.

Edge-centric Partition assign all edges to each subgraph sequentially. When the partition reaches capacity, edges are inserted into a new partition. Every partition has roughly the same number of edges.

BFS Partition Start from a root vertex and travel through the graph in **B**readth **F**irst **S**earch order. Successive vertices are inserted into the subgraph starting from the root vertex. The BFS walk stops when the subgraph reaches its capacity and a new random root vertex is selected for another subgraph.

Multi-Level Partition Partition the graph recursively. Simplified heuristic algorithm and the METIS method are employed.

Given a graph as input, the vertex-centric partition algorithm seeks to find a k-way partition. Each subgraph within the partition has $\lceil \frac{|V|}{k} \rceil$ basic vertices. Edge stream flows into different partitions according to the edge types. EXAMPLE 2 illustrates different types of edge and vertex. As is defined by DEFINITION 3.5.1, the inner vertices and edges are directly included in the partition they belong to. The **cut edge** and two **peripheral vertices** connected by the **cut edge** are added into both partitions for information completeness. A proof for the completeness and result correctness will be provided in SECTION 3.9.

Definition 3.5.1. EDGE AND VERTEX TYPES

*There are three types of edges and two types of vertices. Formally, an edge $e(u,v) \in$ partition P is an **inner edge**, iff all neighbors of two end vertices are in the same partition: $N(e) \in P$. The two end vertices are noted as **inner vertices**. An edge $e(u,v) \in$ partition P is a **peripheral edge**, iff at least one neighboring vertex is in a different partition: $\exists w \in N(e) st. w \neq P$. The end vertex of e that connects to w is called a **peripheral vertex**. An edge $e(u,v)$ is a **cut edge**, iff two end vertices are in different partitions:* WLOG. $(u \in P) \&\& (v \neq P)$. *$u$ and v are peripheral vertices.*

Example 2. *Figure 3.3 provides a simple graph to illustrate different types of graph elements. As is displayed, vertices u_1, u_2, u_3 and v belong to partition P, while vertex*

w is not in P. According to DEFINITION *3.5.1, u_1, u_2 and u_3 are **inner vertices**. Vertex v and w are **peripheral vertices**. Edges $(u_1, u_2), (u_1, u_3)$ are **inner edges** while (v, u_1) and (v, u_3) are **peripheral edges**. Lastly, the edge connecting v and w is a **cut edge**.*

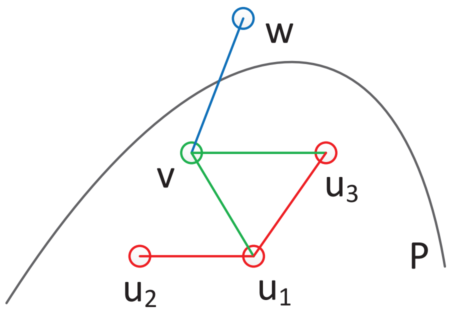


Figure 3.3: Three Edge and Vertex Types

Similarly, given a graph as input, the edge-centric partition algorithm seeks to find a k-way partition. Each subgraph within the partition has $\lceil \frac{|E|}{k} \rceil$ edges. As edge stream flows into different partitions, vertices are marked as inner when they appear for the first time in one subgraph. When any inner vertex is found to appear in a second subgraph, it will be promoted to be a cut vertex. Compared with BFS and Multi-level partition algorithms, the heuristic vertex-centric and edge-centric partition algorithms try to reduce the graph size in the fastest way. The methods for partitioning the vertex (or edge) set into k subsets greatly affects the quality of the algorithm. Since a random assign has led to excessive IO cost, we explore the locality of the graph elements by pre-re-indexing the vertex set.

3.6 Multi-stream Pipelining

After graph partitioning, we explore the parallelism of triangulation on personal computers from the perspectives of application, hybrid system and algorithmic structures. In SECTION 3.3, we have discussed the parallelism of application and GPUs. In this section, we will further exploit the parallelism from multi-core CPU, the algorithmic structure and multi-GPUs.

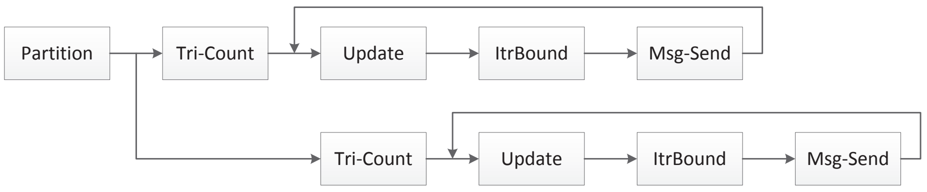


Figure 3.4: Multi-stream Pipelining

Initially, a graph stream flows into main memory as input of the algorithm. After partition, the graph stream is replaced by several sub-streams. As is analyzed in SECTION 3.3, triangulation can be conducted concurrently on these graph sub-streams since a traversal across the graph may start from any subgraphs. The data parallelism among these sub-streams has expressed a demand for parallel operations. In addition, the prevalence of multi-core CPU has promoted the potential physical parallelism in our personal computer systems. In order to explore additional real parallelism, we may try to start several CPU threads at the same time, each of which run on one or several graph sub-streams. By carefully tuning suitable processor affinity, some of these threads can be executed on different CPU cores simultaneously. However, the efficiency and speedup of direct multi-streaming are not as high as our expectation. The problem is that there are some resources which are limited and has become the bottleneck of the system. For example, the L3 cache is shared by all processor cores. In addition, after counting triangles for initial $\widetilde{\lambda}$ values, CPU threads need to send their data onto GPU for iterative bounding. Although modern GPU is capable of executing more than one kernel functions at the same time, the bottleneck still exists since the PCI-E bus has limited bandwidth. Sending all data at the same time results in a traffic jam, slowing down the algorithm.

To solve the problem, we design a multi-stream pipelining mechanism and take advantage of multi-GPUs techniques. Specifically, we first modularize the functional blocks of ALGORITHM 4 and encapsulate them into a sequence of items that can be linked according to their logical order. Graph sub-streams are flowing within the pipes that con-

Algorithm 6: Multi-Stream Pipeline Scheduling

```
1  Scheduler ($P(V', E')$, $tid$)
2      forall the stream str ∈ $P(V', E')$ do
3          if str.thread != $tid$ then
4              $str.thread = tid$ ;
5              forall the segment seg do
6                  while $Serviced[seg]$ == “OCCUPIED” do
7                      THREAD($tid$).wait() ;
8                  $Serviced[seg]$ = “OCCUPIED”;
9                  THREAD($tid$).process($seg$,$str$) ;
10                 $Serviced[seg]$ = “UNOCCUPIED”;
11                 THREAD($tid$).notify() ;
```

nect the functional segments. Then we design a scheduler for the pipelines executed by different threads so as to minimize the idle time and maximize the resources utilization. Furthermore, multi-GPUs as well as PCI-E buses are supported by the multi-streaming pipelining mechanism. FIGURE 3.4 illustrates the multi-streaming pipelining mechanism. After partition, several CPU threads are launched for stream pipelines, each of which runs on one subgraphs. Functional blocks of ALGORITHM 4, Tri-Count, Update, ItrBound and Msg-Send, are encapsulated by segments. Tri-Count is defined as the head of the pipeline while Msg-Send is noted as the tail if the converge requirement is reached.

ALGORITHM 6 illustrates how the scheduler works. A group of global semaphores, noted as “Serviced”, are utilized to flag whether or not the current segment is busy servicing other threads. If the current segment is free, the semaphore for this segment will be locked in Line 8. The thread will then be granted to process the segment on the stream.

In order to facilitate concurrent execution between host and device, we utilize the asynchronous GPU-related function calls. Specifically, data transfer and kernel execution should overlap to support multi-stream pipelining mechanism. In addition, since the kernel execution takes more time than data transfer, multi-kernels need to be executed concurrently as well. Accordingly, ALGORITHM 6 should be modified to support the concurrency of non-conflict segments. Actually, the maximum number of kernel launches that a device can execute in parallel is up to 32. In the context of CUDA environment, we employ routines as displayed in FORMULA 3.1 to combine multi-host-streams with multi-device-streams. Furthermore, whenever there are more than one GPUs in the system, multi-streams can be combined to multi-devices via $cudaSetDevice()$.

$$\begin{aligned} &cudaStreamCreate(\&stream[i]) \\ &cudaMemcpyAsync(iDevPtr, hostPtr, \ldots, stream[i]) \end{aligned} \tag{3.1}$$

Lastly, as graphs grow larger, grid dimension ($\frac{V'}{Db}$) increases and kernel functions take longer execution time. We can chop the vertex set into chunks and utilize multi-stream to explore the parallelism among these chunks. FORMULA 3.2 computes the global indices for threads in this situation.

$$\begin{aligned} &threadID = blockIdx.x * blockDim.x + threadIdx.x \\ &index = chunkIdx * chunkSize + threadID \end{aligned} \tag{3.2}$$

3.7 Dynamic Threading

To explore the performance of triangulation on personal computers, we have tried to exploit many-thread parallelism from hardware and algorithmic perspectives. However, all of these attempts are based on the pre-defined thread configurations. We try to utilize dynamic threading techniques in order to optimize the parallelism from the parallel structure. Dynamic threading enables a running thread to create and synchronize new nested work. Specifically, dynamic threading can easily spawn new threads for nested "for-each" loop. For example, we try to spawn more threads for the "for-each" loop within sub-graphs when initialize the $\widetilde{\lambda}$ values. As is shown in the Line 4 of ALGORITHM 4, several threads can be spawn to represent a chunk of edges. The optimal number of threads depends on the physical processor cores and the multi-stream pipelining mechanism. Similarly, we can also spawn more GPU threads for the loops that compute supporting vertices (Lines 21~24 of ALGORITHM 4) and that bound $\widetilde{\lambda}$ (Lines 25~31). As is illustrated in FIGURE 3.5, GPU spawns more "child" threads to represent the neighbors of "parent" threads.

By employing dynamic threading, parallel triangulation algorithm can be more transparent expressed. Program flow control can be done from within the device, which allows for a hierarchical design of the algorithm.

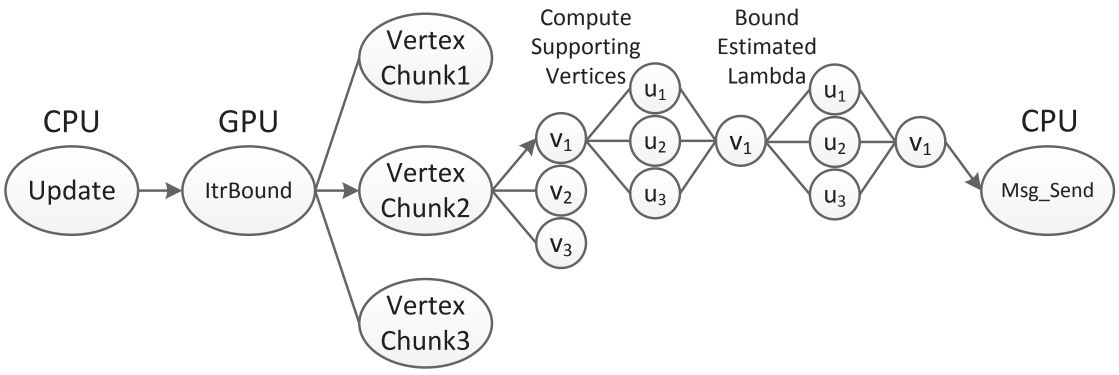
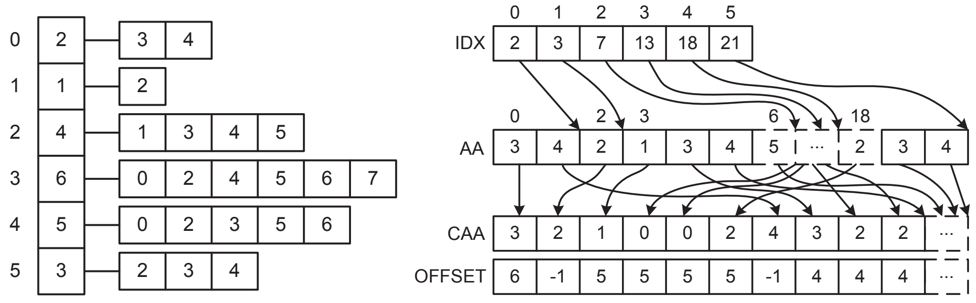


Figure 3.5: GPU Dynamic Threading

3.8 GPU Graph Data Structures

In graph theory, adjacency list and adjacency matrix are data structures for representing graphs. In an adjacency list, each vertex has a list of all other vertices which it connects to. Compared with adjacency matrix, an adjacency list of a sparse graph occupies less space. In most graph data mining algorithms, adjacency list is more efficient for neighborhood access. To facilitate efficient operations on GPU, we compare several data structures for parallel triangulation on GPU. We first adopted the two-array representation of the adjacency list as a baseline graph data structure on the GPU. Then we implemented the column-major adjacency array to enhance memory access efficiency. After that, we propose matrix column-major adjacency array and optimized memory access again. Moreover, we put forward another new data structure named adjacency bitmap to utilize the universal virtual address space.



0 1 2 3 4 5
IDX 2 3 7 13 18 21
AA 3 4 2 1 3 4 5 ··· 2 3 4
0 2 3 6 18
CAA 3 2 1 0 0 2 4 3 2 2 ···
OFFSET 6 -1 5 5 5 5 -1 4 4 4 ···

(a) Adjacency List (b) C(R)-major Adjacency Arrays

Figure 3.6: Row-major and Column-major Adjacency Arrays

Row and Column major Adjacency Arrays

Adjacency list is a compact representation for graphs compared with adjacency matrix. However, the GPU representation of adjacency list is still not so efficient since GPU has limited linear memory space. Therefore, we pack graph adjacency lists row by row into a linear array, with an O(E) space complexity. We named it as graph row-major adjacency array. An auxiliary array with an O(V) space complexity is used to record the ending position of each vertex's adjacency list in the adjacency array. As shown in FIGURE 3.6(b), the IDX and AA arrays are row-major adjacency array representation for the adjacency list in FIGURE 3.6(a). Using the index array, a vertex i can fetch its neighborhood starting from the position IDX[i-1] to IDX[i]-1 in the adjacency array AA.

Considering the CUDA execution model, threads in a block are further divided into 32-thread "*warp*" units, which is the unit of thread scheduling in streaming multiprocessors. The GPU hardware executes an instruction for all threads in the same warp at any given point in time. Accessing a large number of consecutive DRAM locations gets close to the peak global memory bandwidth. When we arrange data to enable all threads

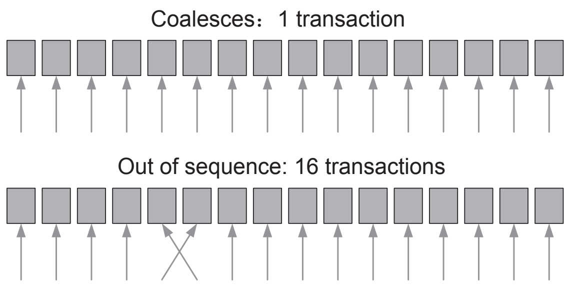


Figure 3.7: Memory Coalesces

in a warp can request consecutive memory addresses, the GPU hardware combines, or coalesces, all of these requests into a consolidated access. Un-coalesce memory access incurs redundant IO cost. For example, in FIGURE 3.6(b), threads 0, 1, 2 and 3 reading position 0, 2, 3 and 7 respectively from array AA bring four separate accesses to the memory instead of one. FIGURE 3.7 displays the benefits of memory coalesces: decreasing memory access latency.

In order to maximize memory coalesces, we attach the neighbors of the vertices in column-wise order. An offset array to record the next neighbor position helps to connect the neighborhood of a vertex effectively. The CAA and OFFSET arrays in FIGURE 3.6(b) are called column-major adjacency array. As is shown in the figure, accessing column-major adjacency array takes advantage of memory coalescing techniques and reduce the IO costs. In the same example, threads 0, 1, 2 and 3 can now read consecutive memory addresses 0, 1, 3 and 3 in array CAA.

Matrix Column-major Adjacency Array

Accessing column-major adjacency array led to high data throughput and low IO costs. However, the imbalance of the neighborhoods still complicates the situation of memory access. In the worst case, excessive control flow divergence to access neighbors from unrelated locations may lead to a huge performance penalty. In addition, reading additional auxiliary array from GPU global memory still wastes data throughput and memory IO-

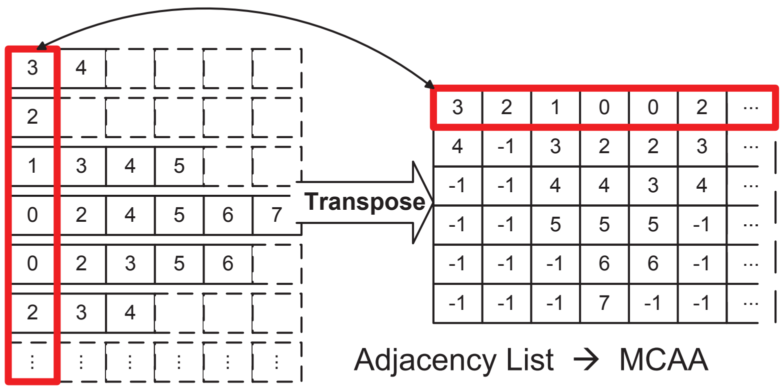


Figure 3.8: Matrix Column-major Adjacency Array

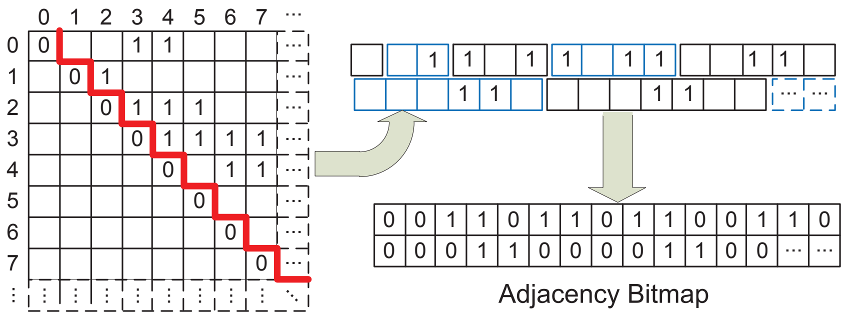


Figure 3.9: Adjacency Bitmap

time. In order to explore multi-thread parallelism, we propose an optimized graph data structure named matrix column-major adjacency array. As shown in FIGURE 3.8, we first expand the adjacency list to a matrix format with placeholders for asymmetric neighborhood. We then transpose the matrix and attach all the rows head to tail into a linear array. Similar to column-major adjacency array, MCAA re-organizes graph adjacency list in a column major style. Although CAA has a more compact data structure, MCAA simplifies the control flow of the algorithm. We define one column as a slice, the size of which equals to $\lceil \frac{|V|}{\text{BLOCKSIZE}} \rceil \times \text{BLOCKSIZE}$.

Graph Adjacency Bitmap

As is discussed above, graph data structures based on adjacency list explore the performance from memory coalescing, compact space and simple control flow. This is true for sparse graphs in general. However, for dense graphs, they do not display enough advantage. Hinted by matrix column-major adjacency array, we propose an adjacency-matrix-based graph data structure named adjacency bitmap. Instead of occupying 4×(—E—+—V—) bytes, adjacency matrix needs only $\frac{|V|^2}{8}$ bytes of linear space. For an undirected graph, only half space of the matrix is needed to save graph information. As shown in FIGURE 3.9, we try to attach the columns of the upper triangle (above the red zig zag line) into a linear array. Similar to MCAA, the length of each column is fixed, nevertheless the column length for vertex varies. FORMULA 3.3 helps to locate the neighboring relationship fast:

$$position = (u < v)\ ?\ \frac{(v-1)v}{2} + u\ :\ \frac{(u-1)u}{2} + v \tag{3.3}$$

Example 3. *In order to decide whether v_1 is connected to v_2, we caculate* $\text{ABMP}[\frac{(2-1)2}{2} + 1] = \text{TRUE}$. *We then say that v_1 does connect to v_2.*

3.9 Result Correctness

For the correctness of parallel triangulation, we may prove it in two steps. For the first step, we proved the correctness of λ bounded in parallel theoretically. For the second step, we compared $\lambda(\text{G})$ computed with that of sequential triangulation.

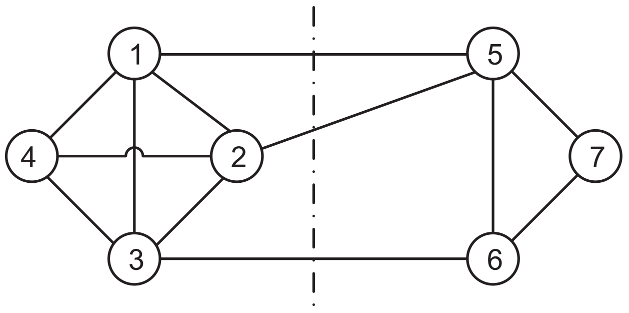
Theorem 3.9.1. Result Correctness

Triangulation on subgraphs generates partial bounding for each edge. Combining the partial results can return a correct estimation of λ value for each edge.

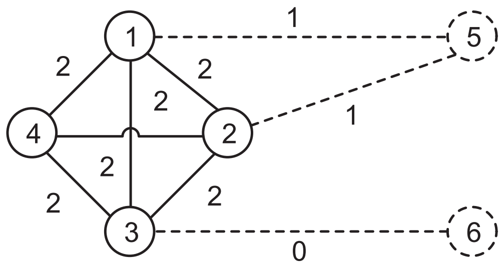
Proof. Partitioning algorithm reads a stream of edges and assigns each edge $e(u, v)$ to a subgraph according to the index of the two end vertices. A cut-edge is marked when its two ends belong to two different subgraphs. For all internal edges, triangulation within a subgraph generates the same result with that of the original graph. For those cut-edges, we need to consider all triangles that they belong to in both subgraphs. Let $e(u,v)$ be one peripheral edge within subgraph S_1 and vertex w belongs to a different subgraph S_2. The two cut-edges (u,w) and (v,w) will be streamed in individually. When a cut-edge, say (u,w) WLOG., is read, w will be added into S_1 and u is included into S_2. Moreover, (u,w) will also be added into S_1 and S_2. Triangulation will be conducted to (u,w) in both subgraphs.

For peripheral edges, they can be part of triangles in two situations. One type of triangle contains cut-edge while the other does not. All triangles that do not contain any cut edge will be counted within each subgraph. Since all cut-edges are added into both subgraphs and the peripheral edge $e(u,v)$ will only be in one subgraph S_1, the triangle containing cut-edge will appear in only one subgraph. This means no triangle will be missed. Even though the triangulation in subgraph S_2 may result in a partial value for cut-edges, the merging it with that from S_1 will maintain a correct result. □

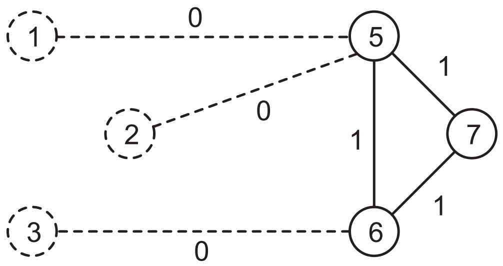
Example 4. *Figure 3.10 displays an example for our result correctness proving. The graph* G *is partitioned into two subgraphs* $\mathrm{S}_1\{1,2,3,4\}$ *and* $\mathrm{S}_2\{5,6,7\}$ *in Figure 3.10(a). Then we apply triangulation on individual subgraph* S_1 *and* S_2 *as shown in Figure 3.10(b) and 3.10(c). In* S_1*, vertices 5 and 6 are included as well as three cut-edges (1,5),(2,5) and (3,6). They are displayed as dotted circles and lines. Triangle* $\triangle(1,2,5)$ *does not contain two cut edges. We can see all the edges have correct* λ *values in subgraph* S_1 *after triangulation. In* S_2*, vertices 1, 2 and 3 as well as cut-edges (1,5),(2,5) and (3,6) are included in partitioning. After triangulation, the* λ *values for edges (1,5) and (2,5) are partial. By merging,* $\lambda(1,5) = \max(\lambda_{\mathrm{S}_1}(1,5), \lambda_{\mathrm{S}_2}(1,5)) = 1$ *and* $\lambda(2,5) = \max(\lambda_{\mathrm{S}_1}(2,5), \lambda_{\mathrm{S}_2}(2,5)) = 1$.



(a) Graph G Partitioning



(b) S_1 Triangulation



(c) S_2 Triangulation

Figure 3.10: Result Correctness

3.10 Experiments

In this section, we study the parallelization of iterative triangulation algorithm running on heterogeneous personal computer systems. As an accelerated application, parallel triangulation is designed for solving the two main bottlenecks previously noticed in personal computer systems. For the first step, we look into the performance of parallel triangulation. We compare the speedup and efficiency when parallel triangulation is executed in different modes. We compare several disk-based partitioning algorithms and discuss multi-stream pipelining techniques. Additionally, we study several new graph data structures on GPU. Finally, we try to discuss the influence of GPU execution configurations.

We have conducted our experiments on three GPU accelerated personal computer systems. They are a DELL PC equipped with an NVIDIA GeForce 9400 GT graphics card, an ACPI x64-based PC accompanied with an NVIDIA GeForce GT 520 graphics card and another ACPI x64-based PC equipped with an NVIDIA GeForce GT 330 graphics card. The price of the simple GeForce 9400 GT and GeForce GT 520 graphics card are less than 50 US dollars respectively while the price of GeForce GT 330 graphics card is around 100 US dollars. We present all major technical specifications of the three platforms in Table 3.1.

Platform	**CPU**	**Main Memory**	**GPU**	**GPU Memory**	**Compute Capability**	**GPU Price**
DELL Opti-755 Geforce 9400 GT	2 cores 2.33 GHz	3.3 GB DDR2	1×16 cores 550 MHz	512 MB 400 MHz	1.1	< $50
ACPI x64-based Geforce GT 520	4 cores 3.40 GHz	8 GB DDR2	1×48 cores 1.62 GHz	993 MB 535 MHz	2.1	< $50
ACPI x64-based Geforce GT 330	4 cores 3.40 GHz	8 GB DDR2	12×8 cores 1.34 GHz	1 GB 790 MHz	1.2	< $100

Table 3.1: Experimental Platforms

We employ three main datasets in our experiments. Flickr dataset is derived from the well-known photo sharing network Flickr with 1,715,255 vertices and 22,613,982 edges. Each vertex represents a person and the edge denotes two people share photos with each other. Protein Protein Interaction (PPI) dataset contains 17203 interactions among 4930 proteins. Netflix is an American provider of on-demand internet streaming media. There are 480,000 customers and 17,000 movies in the dataset.

Table 3.2 lists the experimental parameters and their corresponding experimental options (default values are marked in bold). We partition the graph containing $|V|$ vertices and $|E|$ edges into *parNum* subgraphs using *parMtd* method. The size of the buffer in the memory is set to be *bufSz*. As a "pay as you go" strategy, we can also choose to exit the outer loop after *maxItrO* iterations and inner loop after *maxItrI* iterations in our parallel triangulation algorithm. Moreover, as a systematic solution, we can choose to run the iterative triangulation operator in *mineMode* mining mode. Given "hybrid" mode is selected, there are four options for *graphDS* graph data structure.

Parameter	**Experimental Option**
$\lvert V\rvert$: Number of vertices	[1715255]
$\lvert E\rvert$: Number of edges	[22613982]
parNum: Number of partitions	[50, 100, 200, 250, 300, 350, **400**]
parMtd: Partition Method	[**vertex-centric**, edge-centric, BFS, multilevel]
bufSz: Buffer Size	[4KB, **8KB**, 16KB, 32KB, 64KB]
maxItrO: Max Outer Iterations #.	[3, 4, **5**, 6, 10]
maxItrI: Max Inner Iterations #.	[5, **10**, 15, 20, 25]
mineMode: Mining Mode	[Host, Device, **Hybrid(SHSD)**]
graphDS: Graph Data Structure	[AA, CAA, **MCAA**, ABMP]

Table 3.2: Parameter Table

3.10.1 Performance Evaluation

In this section, we study the performance of our parallel triangulation algorithm. As an economic solution for speeding up the triangulation on large graphs, we implement our algorithm across gpu-accelerated hybrid architecture.

Platform	**Mode**	**Partition**	**Tri-Count**	**ItrBound**	**Total**
Work Station 128GB RAM	-	-	4.88 hrs	69.3 hrs	74.18 hrs
Dell 2Cores 3.3GB RAM	Host	4.67 hrs	985.07 sec	207.99 hrs	213.04 hrs
ACPI 4Cores 8GB RAM	Host	3.91 hrs	872.65 sec	145.59 hrs	149.85 hrs
Dell 2Cores GF9400,16Cores	Hybrid	4.62 hrs	998.37 sec	26.46 hrs	31.46 hrs
ACPI 4Cores GF520,48Cores	Hybrid	3.9 hrs	894.16 sec	23.1 hrs	27.35 hrs
ACPI 4Cores GF330,96Cores	Hybrid	3.92 hrs	894.16 sec	17.3 hrs	21.46 hrs

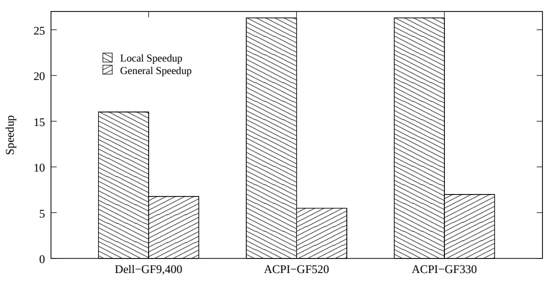
Table 3.3: Response Time for Each Component

As stated in section 3.6, parallel triangulation algorithm can be expressed as a series of functional blocks. TABLE 3.3 lists the running time for major functional components in ALGORITHM 4. As a baseline algorithm, sequential iterative triangulation is memory-bound. For massive graphs, sequential iterative triangulation applies only on work stations with large memory. We compare the performance when the algorithm is running on the DELL desktop and the ACPI system. In addition, we also compare the performance of the parallel algorithm accelerated by different GPUs.

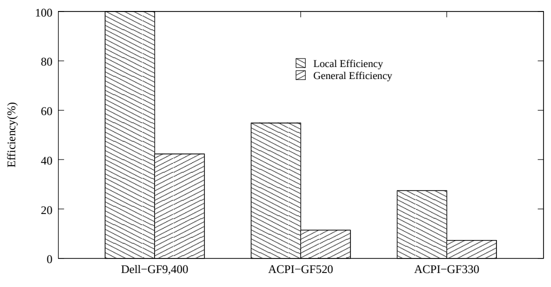
As shown in TABLE 3.3, sequential iterative triangulation and parallel triangulation running in "host" mode take much more time than parallel triangulation accelerated by GPU. With more processing cores, GeForece 330 GT gpu provides more computing power for ACPI system. Although the local speedup S_l defined in DEFINITION 2.1.5 is difficult to measure, however, it is possible to estimate its value using an approximate formula. Let N be the processor number and $\overline{N}(e)$ be the average neighborhood size. $S_l = \min(N, \overline{N}(e)) = \min(N, \frac{2\times|E|}{|V|}) = \min(N, 26.3)$. Therefore, the S_l for DELL desktop is 16, and 26.3 for the other 2 personal computers. Similarly, the global speedup S_g can be calculated for a specific platform. $S_g = \frac{T_1(\Delta(G))}{T_N(\Delta(G))} = \frac{213.04}{31.46} = 6.77$ for the DELL system and $S_g = \frac{149.85}{21.46} = 6.98$ for the ACPI system equipped with GeForce GT 330 GPU. The local efficiency $E_l = \frac{S_l}{N} = \frac{\min(N, 26.3)}{N}$. Therefore, the local efficiency for DELL desktop is 1, since it takes full use of all its processors. And the local efficiency for the ACPI system equipped with GeForce GT 520 GPU is $E_l = \frac{\min(N, 26.3)}{N} = \frac{26.3}{48} = 54.8\%$. The local efficiency for ACPI system equipped with GeForce GT 330 GPU is $E_l = \frac{26.3}{96} = 27.4\%$. The general efficiency $E_g = \frac{S_g}{N}$ differs for different platforms. $E_g = \frac{6.77}{16} = 42.3\%$ for the DELL desktop and $E_g = \frac{6.98}{96} = 7.27\%$ for the ACPI system equipped with GeForce GT 330 GPU. The general efficiency $E_g = \frac{149.85}{27.35\times 48} = 11.4\%$ for the ACPI system equipped with GeForce GT 520 GPU.

Figure 3.11 compares the speedups and efficiencies among different platforms. We can observe that general speedup S_g is usually smaller than local speedup S_l, because there are some portions of the algorithm that cannot be parallelized. The two ACPI systems have lower local efficiencies because they cannot utilize all of the parallelism provided by the hardware. Even with more processors, the system has shown a lower efficiency. The system cannot make full use of all the additional computing power.

We also study the influence of two iteration parameters, *maxItrO* and *maxItrI*. When only one parameter is raised, the running time of the algorithm rises as well. As shown

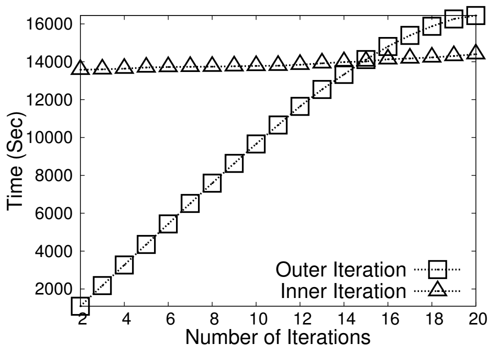


(a) Speedup

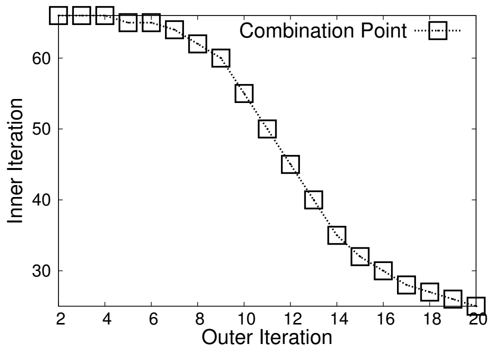


(b) Efficiency

Figure 3.11: System Performance



(a)



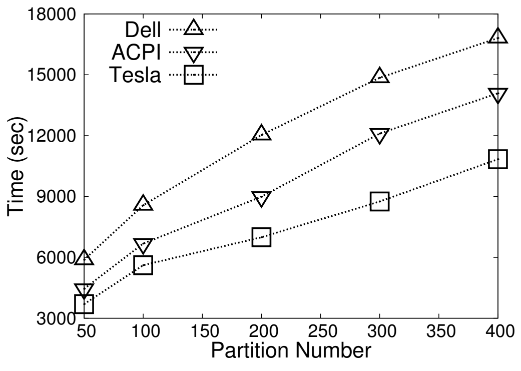
(b)

Figure 3.12: Iteration Parameters Study

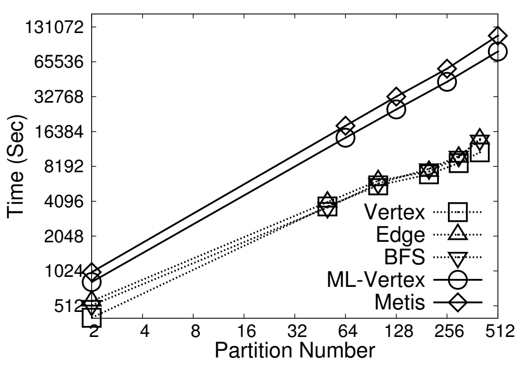
in FIGURE 3.12(a), the maximal outer iteration number *maxItrO*, has higher influence over the system than the maximal inner one *maxItrI*. When *maxItrO* adds to more than 16 iterations, the increasing of the running time becomes slower, since some subgraphs have converged. An additional inner iteration runs faster with powerful computability provided by GPU. FIGURE 3.12(b) depicts a combination point study which means the study of a pair of parameters that converges the iterations. When there are $15 \sim 21$ outer iterations, small inner iteration number is observed.

3.10.2 Partitioning Algorithms

As a streaming solution for processing large graphs, we study our partitioning algorithms in various system environment. As discussed in Section 3.5, we employ five partitioning streaming partitioning algorithms. They are vertex partitioning, edge partitioning, BFS partitioning, multi-level simple partitioning and metis partitioning. Figure 3.13 presents the performance comparison of the five algorithms in two low-end systems as well as system with higher computability.



(a) Computability



(b) Partition Strateges

Figure 3.13: Partitioning Performance

To study the effect of computability on partitioning performance, we test our partitioning algorithms as partition number increases. In figure 3.13(a), the running time of our partitioning algorithm increases steadily, with an increasing partition number. For example, the running time increases from about 1.5 hours to 4.5 hours in the server. Since our partitioning algorithms handle graph streams from the external storage to main memory, we only need to consider the CPU computability and main memory size when comparing different computer platforms. As is noted from table 3.1, ACPI system has double CPU cores and main memory size than DELL one. Therefore, we can see a proportional decrease in the running time of the partitioning algorithm running on different platforms. In figure 3.13(b), we compare the performance of our five partitioning algorithms when partition number increases from 2 to 512. The x-axe and y-axe are both in log-scale, and five lines are almost linear. We can easily observe that the two recursive(multi-level) algorithms run longer than the other three. This is because the recursive partitioning algorithms scan the whole graph for each level, which increases the IO time. Running with the simplest heuristic logic and scanning the graph for only one pass, vertex partitioning is the fastest among all five partitioning algorithms.

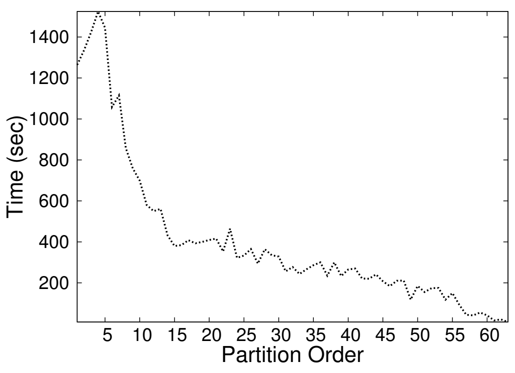


Figure 3.14: Partition Order

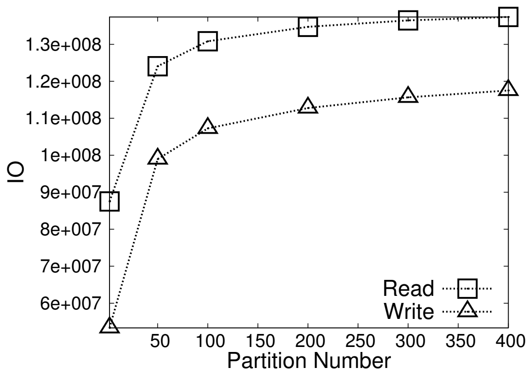


Figure 3.15: Partitioning I/O

Comparing to BFS partitioning, recursive(multi-level) partitioning travels the graph in DFS style. Figure 3.14 records the partitioning time for subgraphs and the doted-line forms a power-law like shape. This reflects the flickr data is a power-law graph and vertices are ordered according to their neighborhood size.

FIGURE 3.15 depicts the IO cost of vertex partitioning. Since we use buffer to cache the vertices and edges before they are written to a specific subgraph file, the output operations are less than the input ones. For a better observation, the starting point of the x-axe is two in the figure.

3.10.3 Graph Data Facilities

In this section, we study the graph data facilities namely the cut-edge message files and the graph data structures on GPU. On one hand, since the cut-edge message files spread the messages among subgraphs, our algorithm uses them to communicate. Therefore, the usage of the buffer affects the IO performance of the algorithm. In our experiment, we vary the buffer size and observe the changes of the IO in our program.

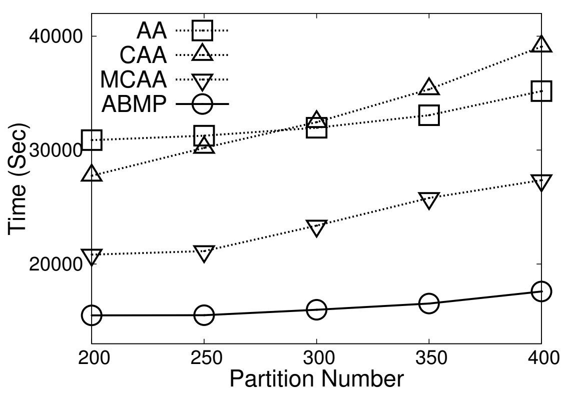


Figure 3.16: GPU Graph DS

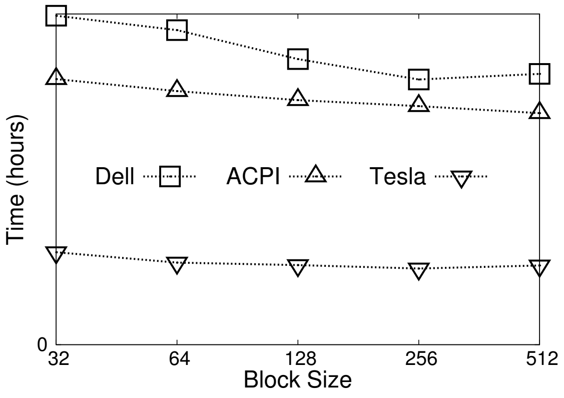


Figure 3.17: Varying Block Size

On the other hand, we test the running performance about the graph data structures on GPU. In this chapter, several novel graph data structures are proposed on GPU. Figure 3.16 compares the running time when different graph data structures are used in our two-level iterative triangulation algorithm. We can observe that the running time of the algorithms increases when the flickr graph is divided into more subgraphs. Obviously, MCAA (matrix column major adjacency array) and ABMP (adjacency bitmap) have better performance than AA (adjacency array) and column major adjacency array. Since CAA is used for accelerating batch data transfer, larger subgraphs from a smaller partitioning number benefits more from the data structure. In the figure, when partitioning number is smaller than 280, program using CAA runs faster than AA. Because CAA counts on vertex locality, a larger graph portion makes an advantage and saves additional time to load the auxiliary array from the global memory. Figure 3.18 displays the speedups of the three novel graph data structures when comparing with the adjacency array. As we can observe, ABMP has a stable speedup while CAA and MCAA have decreasing ones. This reflects the fact that the graph size has few influence over the adjacency bitmap data structure.

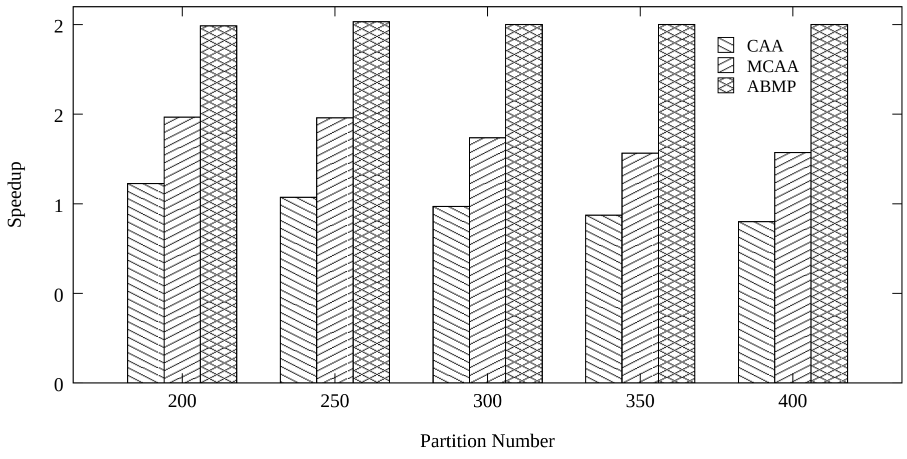


Figure 3.18: GPU Graph DS Speedups

3.10.4 GPU Execution Configurations

GPU execution configuration is an important factor for any CUDA-based program. As components to GPU execution, block size Db, grid size Dg and shared-memory size Ds are also parameters to our program. Since Dg can be decided by the data size and Db, we only need to consider two additional parameters: Db and Ds. Varying such parameters can affect system performance. For example, figure 3.17 compares the performance of our algorithm on different platforms with various block size Db. We can observe that Dell and Tesla get an optimal performance while Db is set to be 256. And ACPI system reaches its optimal value when Db equals to 512. The difference exists because different GPUs have different compute capabilities. ACPI has a graphics card that has more registers and can accommodate more threads and warps.

3.11 Summary

In this chapter, we parallelize triangulation on heterogeneous computing systems. Lacking computing power and limited memory space are two main bottlenecks for large graph triangulation. We presented a streaming partitioning strategy to divide the graph so that each subgraph can be read into memory. A parallel algorithm was then proposed for triangulation on GPU-accelerated heterogeneous platform. Moreover, an efficient message spreading mechanism was then designed to facilitate inter-partition communications. To further exploit nested parallelism of the algorithms, we propose dynamic threading and streaming pipelining approach. In addition, several novel GPU graph data structures were designed to enhance GPU processing efficiency. Extensive experiments showed that our solution had achieved a notable speedup.

Chapter 4

SIGPS: Synchronous Iterative GPU-accelerated Graph Processing System

In this chapter, we propose SIGPS (Synchronous Iterative gpu-accelerated Graph Processing System), a generic graph processing system built on many-core GPU platform. This is a general solution provided for graph processing on GPU-accelerated PCs. A Pregel-like BSP-based computation model is designed in combination with the state-of-the-art GPU high performance computing techniques. Emulating shared memory is used to assist fast communication among concurrent GPU threads. User-friendly high level C++ APIs are provided. Programmers can implement their algorithms using the generic interface and code in a simple sequential style. Lastly, automatic GPU execution configuration and dynamic thread allocation are supported by SIGPS.

4.1 Problem Statement and Design Purpose

In data-centric settings, graphs can be highly unstructured. Algorithms on these graphs are particularly difficult to parallelize on distributed memory machines. Actually, these algorithms tend to exhibit a high degree of fine-grained parallelism. They can be mapped more pertinently to massively multi-threading and shared memory paradigm. The execution model bases on the availability of a large number of threads to keep the processors busy. So as to implement high performance parallel graph algorithms, coarse-grained to fine-grained parallelism need to be addressed. Memory contention is another critical issue to be considered. Simultaneously accessing the same memory address may incur correction problem, while coalescing memory access can enhance thread concurrency. Load balancing improves performance by reassigning tasks among threads.

In consideration of the flexibility, extensibility, portability and maintainability properties, a generic graph processing system on multi-threaded shared-memory paradigm is more useful for users when comparing to existing tools such as MapReduce, Parallel Boost Graph Library [16] and Multi-Threaded Graph Library [31]. Similar to MapReduce, the generic graph processing system should provide easy-to-use application programming interfaces. By simply re-writing the abstract class or implementing the interface, researchers can easily program their algorithms. Such kind of system enables users to focus on the core functionality of the specific problem, instead of devoting to handling complicated low-level system programming issues. The programming model for such system has a shallow learning curve.

Specifically, there are three main design aims for SIGPS:

1. To implement a Pregel-like graph processing system.

2. To utilize GPGPU techniques for graph processing on shared memory platform.

3. To provide a as simple as possible generic API for users to implement their algo-

rithms.

The rest of this chapter is organized as follows. In Section 4.2, the Pregel-like BSP-based computation model and the system state machine are displayed. Besides, the system architecture overview is also provided in this section. Section 4.3 gives an overall description of the system and introduces its main components. Auxiliary components of the system are then discussed in Section 4.4. Section 4.5 presents automatic execution configuration and dynamic thread allocation. We will study three cases in Section 4.6. The generic vertex APIs usage demonstration are provided in Section 4.7. Extensive experiments in Section 4.8 demonstrate that SIGPS is applicably generic and efficient. We finally conclude in Section 4.9.

4.2 Computation Model and System Overview

Similar to google's Pregel, SIGPS bases itself on a vertex centric approach and the Bulk Synchronous Parallel (BSP) computing model. The processing of a graph consists of a series of iterations (supersteps). Each vertex is represented by one light-weight GPU thread. Users utilize the generic API provided by the system and implement their algorithms into a "compute" member function of the vertex. The system then launches a batch of concurrent threads over a large graph data set on a local shared-memory machine.

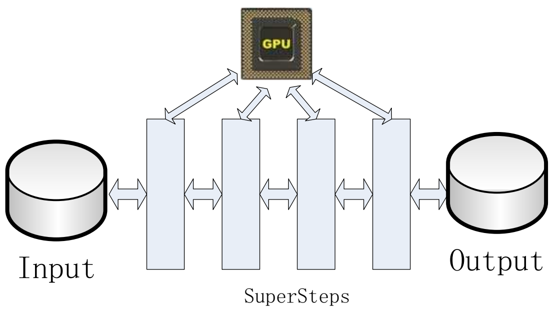


Figure 4.1: SIGPS Computation Model

Figure 4.1 illustrates an overview of the SIGPS computing model. Graph behaves as a data flow running between input and output. The computation and communication of the graph elements are divided into subsequent stages. These stages are also regarded as *SuperSteps*. During each superstep, graph data is sent to be processed by many-core GPU. Between subsequent supersteps, a physical barrier is set to force synchronization among threads.

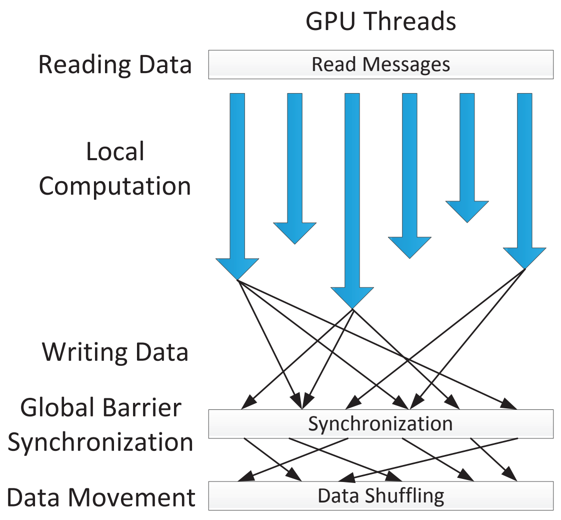


Figure 4.2: GBSP Model

The GPU Bulk Synchronous Parallel (GBSP) model displayed in Figure 4.2 consists of three main components:

computation: User-defined "compute" function is executed by each vertex separately.

communication: Vertex reads messages from its neighbors, encapsulates its value into messages and sends back to neighbors.

synchronization: All vertices wait for each other at the end of one iteration.

We utilize the state-of-the-art many-core GPGPU techniques to implement this processing model. The compute-function is written as a GPU kernel function that is invoked by CPU and executed on GPU. Computation is done locally by a batch of GPU threads on top of the CUDA programming model. Message transmission and data shuffling are conducted within the GPU memory, PCI-Express system buses and main memory. In the beginning of one superstep stage, each vertex read messages received to update its local vertex value. A local barrier is set to make sure that all updates are applied. After local computation, each vertex sends its new-computed value to all its neighbors, where another global barrier is needed. Synchronization is forced when GPU kernel functions are invoked and terminated under the orders sent from CPU to GPU.

The general system architecture of SIGPS applies a master-worker pattern. A master module corresponds to the main thread of the system. The master divides the graph into several sub-graphs and dispatches them to several worker threads. The workers are initialized by the CPU threads. Then these workers make a spinning and wait for signals from the master. A batch of GPU threads representing vertices within the subgraph are then launched by each worker. As a system design for scalable heterogeneous computing platform, SIGPS also supports multi-GPUs and distributed computing. A module named worker manager is employed between the master and the workers. Each worker manager corresponds to one GPU and is registered with the master. After retrieving tasks and data from the master, the worker manager initializes the sub-tasks/sub-graphs. The worker manager module acts as an interface layer between CPU and GPU.

Figure 4.3 shows a hierarchical overview of the SIGPS model. A master takes charge of several worker managers, each of which corresponds to one GPU. The worker manager initializes a group of workers by assigning each of them a set of graph partitions. On receiving signals from the worker manager, the workers iteratively loads assigned graph partitions onto GPU memory and launches kernel functions on graphics parallel processing units.

Similar to Pregel, SIGPS is also a vertex-centric model. A vertex compute-function is executed by one GPU thread which represents one vertex. Inside the vertex compute-

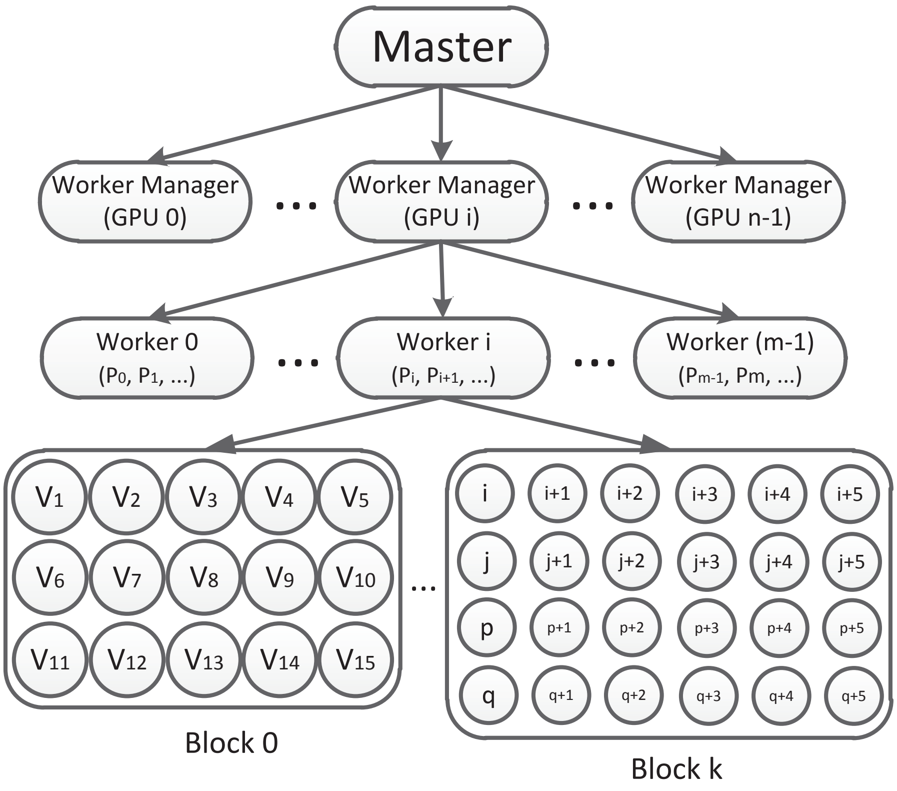

Master
Worker Manager (GPU 0)
Worker Manager (GPU i)
Worker Manager (GPU n-1)
Worker 0 (P_0, P_1, ...)
Worker i (P_i, P_{i+1}, ...)
Worker (m-1) (P_{m-1}, Pm, ...)
V_1 V_2 V_3 V_4 V_5
V_6 V_7 V_8 V_9 V_{10}
V_{11} V_{12} V_{13} V_{14} V_{15}
i i+1 i+2 i+3 i+4 i+5
j j+1 j+2 j+3 j+4 j+5
p p+1 p+2 p+3 p+4 p+5
q q+1 q+2 q+3 q+4 q+5
Block 0
Block k

Figure 4.3: SIGPS Architecture

function, the state of a vertex or an edge can be altered. Each vertex may receive messages from its neighbors, does some computation and spreads out the result via its outgoing edges.

Different from the distributed infrastructure of Google's Pregel, SIGPS utilizes the state-of-the-art GPU hardware and drives the graph computation using blocks of GPU light-wight threads. The algorithm within the vertex compute-function terminates when all vertices *vote to halt*. Figure 4.4 illustrates the state machine of SIGPS. In the first *SuperStep*, all vertices represented by threads in a block are in active state; all blocks of threads then participate in the computation of the algorithm; Some vertex may vote to halt and pause its corresponding thread; As an inactive vertex receives some external message, it can be re-activated and return to the active state. Only when all threads within the block are inactive, this *SuperStep* can declare to be suspended by the WorkerManager.

While all WorkerManagers agree on an end to the *SuperStep*, the algorithm may move to the next one. This is called an explicit global synchronization operation in the model.

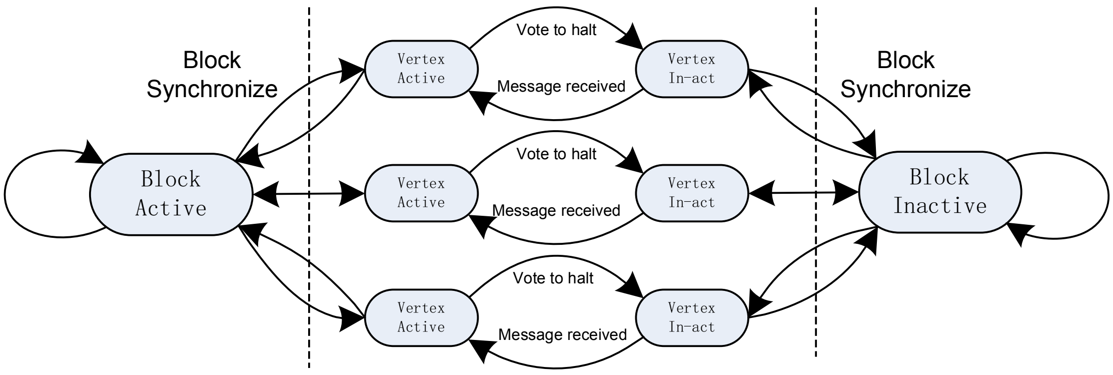


Figure 4.4: Block State Machine

Instead of using the message passing model of Google's Pregel, we chose to apply an emulating shared memory model for communications between the successive *SuperSteps*. There are three considerations when we design SIGPS's communication mechanism. First, GPU's thread model is built on top of its memory. Since we utilize GPU's thread model for parallelizing graph computation, we would prefer to use its hierarchical memory for communication. Second, the motivation of designing SIGPS is to provide a generic and powerful graph processing system for ordinary researchers who are incapable of accessing a Pregel-like high performance computing system. Last but not least, SIGPS has a high level computation model in which the message is just an abstract interface for communications. Message passing and emulating shared memory are two specific communication implementation strategies. We can extend our emulating shared memory model to distributed environment when a high performance GPU cluster is considered in the future.

Before a batch of threads are launched on the GPU, a region of GPU memory is initialized as a message center. Threads across the blocks use this region of memory as a base for message exchange. Utilizing the characteristic of GPU memory hierarchy, processing device function shuffles the data for threads. For intra-block communication, message center takes advantage of the multi-level cache and on-chip shared memory to quickly exchange data among threads. For inter-block communication, message center employs optimized shuffling algorithms among threads. As a data consistency requirement, synchronization is also provided in the two levels of thread hierarchy, intra-block and inter-block.

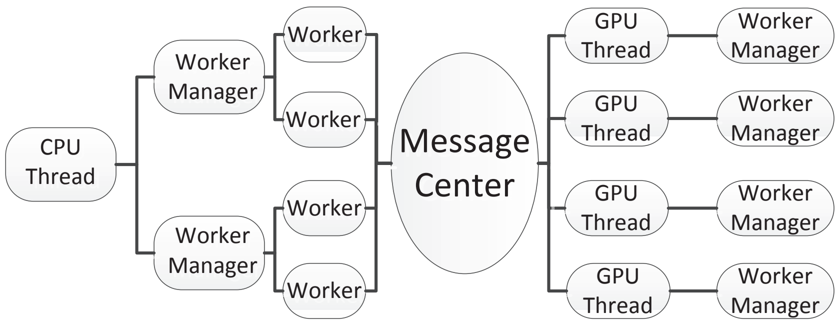


Figure 4.5: System Overview

Figure 4.5 displays the system architecture overview. A CPU thread invokes several Worker Managers, each of which launches a number of GPU threads. The user-defined compute function is then executed by GPU worker threads concurrently as soon as the data flow arrives. After computation, all workers send their results to the message center, in which messages are processed and grouped by GPU threads before posting to their destinations. And starting from the second *SuperStep*, each Worker Manager will receive messages and then assign them to the corresponding worker threads. In a word, SIGPS computation model uses iterations over *SuperSteps* to avoid the programming complexity of graph algorithms and associated serialization overhead. The application of GPU threads and emulating shared memory model has successfully eliminated a large amount of remote communication overhead.

As we know, modern GPU has evolved into many heavily multi-threaded-core processors. There are three levels of thread hierarchy when the graph model is mapped onto

GPU SIMT architecture. At the first level, vertices from SIGPS are dynamically grouped into warps and scheduled to run on SIMD hardware at runtime. The warp is transparent to the users. At the second level, bundles of vertex warps are grouped into a block, within which vertices can synchronize via local barriers specified by the user. Vertices within the same block are always assigned to the same SM [1]. We say these vertices are strong-correlative. At the third level, multiple blocks are further grouped into a grid. Vertices from different blocks are regarded as weak-correlative. Vertex blocks can be executed in scheduled order. In one word, finer thread-level parallelism and synchronization provide users more flexibility to implement their algorithms using simple generic APIs.

4.3 Overall Description and System Main Components

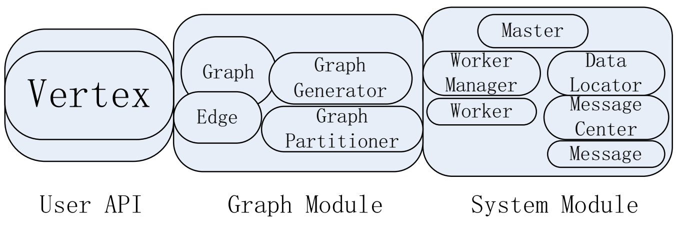


Figure 4.6: Software Architecture

SIGPS consists of three main modules, namely User API module, Graph module, and System module. Figure 4.6 displays the software architecture of SIGPS. From left to right, the degrees of the functional transparency and user relativity decrease. Users can access to the generic APIs provided by the vertex class. They invoke a set of vertex member functions to implement their algorithms into the “compute” function. Graph module contains several related classes for graph processing. Users cannot directly use functions within these classes. However, they may use related vertex APIs to interact with

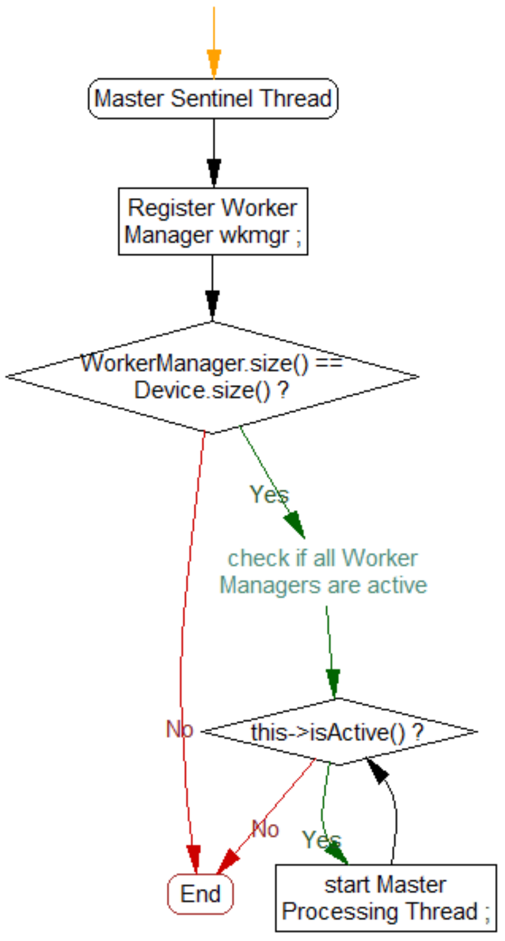
[1]Streaming Multiprocessors

these classes when composing their own algorithms. System module is the functional skeleton for SIGPS. Master, worker manager and worker define the SIGPS architecture. Data locator and message center relate to the communication, one of the most important components of SIGPS.

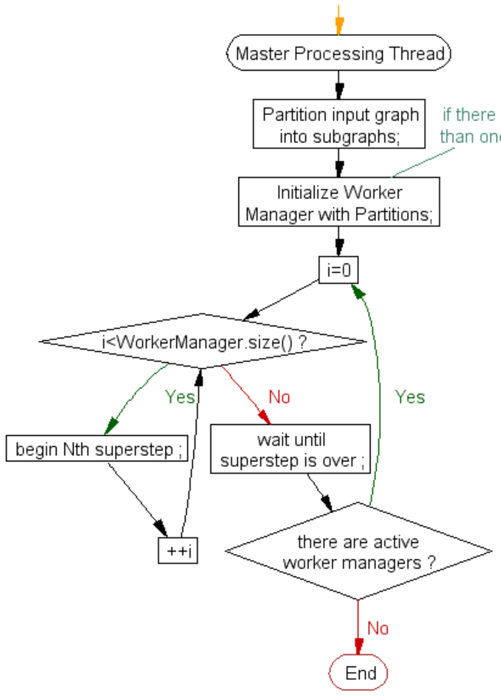
In this section, we introduce the architecture of the main components of SIGPS. There are three major modules in SIGPS, each of which contains several classes. The user API module has only two “Vertex” classes. One is defined for CPU and the other is implemented in the GPU memory and designed for GPU operations. The graph module contains the “Edge” class, “GraphGenerator” class, “GraphPartition” class and “GraphPartitioner” class. The system module consists of Master, Worker, Communication and Utility submodules. Specifically, the Master submodule has “ManagerToMaster” abstract class, “IdManagerMap” and “MasterImpl” classes. The Worker submodule includes the “Worker” class, “WorkerManager” abstract class and “WorkerManagerImpl” class. The Communication mechanism is composed of a “Communicator” class, a “DataLocator” class, “Message” class and “MessageSpooler” abstract class. The Utility includes “Logger”, “lock” and “FaultDetector”. In the following subsections, we will introduce the architecture of those important classes such as “Master”, “Worker Manager”, “Worker”, “Vertex”, “Partitioner” and “Communicator”.

4.3.1 Architecture of Master

Besides maintaining a list of worker managers, there are two threads for the master class running on CPU, the Sentinel and Processing threads. As is illustrated in Figure 4.7(a), the Master Sentinel acts as a guard to poll all the active worker managers. The Sentinel is triggered when a new worker manager is registered. While all worker managers are ready, the Sentinel launches the Master Processing Thread. The architecture of Master Processing Thread is shown in Figure 4.7(b). The Processing Thread partitions the graph


Master Sentinel Thread
Register Worker Manager wkmgr ;
WorkerManager.size() == Device.size() ?
Yes
check if all Worker Managers are active
this->isActive() ?
No
No
Yes
End
start Master Processing Thread ;

(a) Master Sentinel Thread


Master Processing Thread
Partition input graph into subgraphs;
if there are more than one devices
Initialize Worker Manager with Partitions;
i=0
i<WorkerManager.size() ?
Yes
No
Yes
begin Nth superstep ;
wait until superstep is over ;
++i
there are active worker managers ?
No
End

(b) Master Processing Thread

Figure 4.7: Master Architecture

into subgraphs according to the number of machines and the number of GPUs each machine has. Then the Processing Thread initializes the worker managers and passes the subgraphs to them. If there is only one machine and one GPU in the physical system, the master launches only one worker manager using the original graph. After that, the Processing Thread starts to execute supersteps only if there is any registered worker manager is active.

The main functions of the master are listed as follows:

- Register Worker Managers.

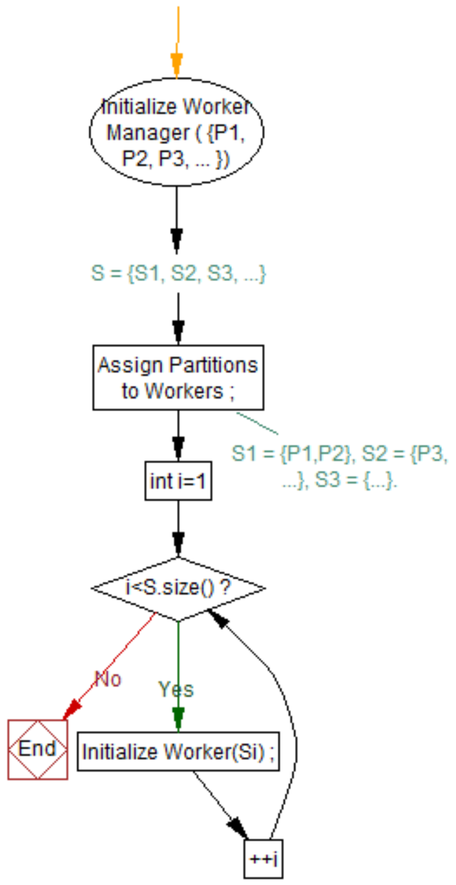
- Poll Worker Managers, if all are activated, start to process the graph.
- Signal to stop active Worker Managers.
- Partition the graph according to device counts.
- Initialize Worker Managers.
- Start supersteps to process the graph.
- Output the results.

4.3.2 Architecture of Worker Manager

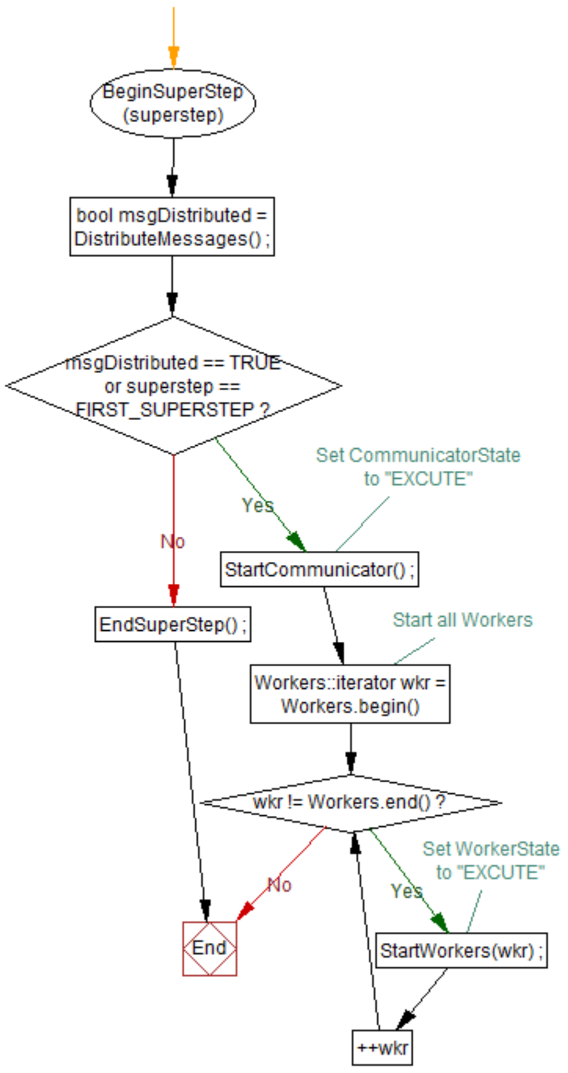
A Worker Manager acts as an interface between CPU and GPU that handles all the tasks assigned to one GPU. It takes charge of several workers and a batch of GPU threads. As displayed in Figure 4.8(a), a worker manager is initialized by the master thread with a set of assigned partitions. The worker manager further divides the assigned partitions among its workers and then initializes these workers. As shown in the second step of Figure 4.7(b), the worker managers should respond to the master and start supersteps to process the subgraphs. In Figure 4.8(b), a worker manager distributes messages to all the destination vertices at the beginning of a superstep. From the perspective of vertex, it receives/reads the messages in this stage. If it is in the first superstep or all the message received by this worker manager from the last superstep, the worker manager signals the communicator and the workers start to execute.

The main functions of the worker manager are listed as follows:

- Initialize worker manager.
- Initialize workers and assign partitions.


Initialize Worker Manager ({P1, P2, P3, ... })
S = {S1, S2, S3, ...}
Assign Partitions to Workers ;
S1 = {P1,P2}, S2 = {P3, ...}, S3 = {...}.
int i=1
i<S.size() ?
No
Yes
End
Initialize Worker(Si) ;
++i

(a) Initialize Worker Manager


BeginSuperStep (superstep)
bool msgDistributed = DistributeMessages() ;
msgDistributed == TRUE or superstep == FIRST_SUPERSTEP ?
Set CommunicatorState to "EXCUTE"
Yes
No
StartCommunicator() ;
EndSuperStep() ;
Start all Workers
Workers::iterator wkr = Workers.begin()
wkr != Workers.end() ?
Set WorkerState to "EXCUTE"
No
Yes
End
StartWorkers(wkr) ;
++wkr

(b) Begin Superstep

Figure 4.8: Worker Manager Architecture

- Initialize communicator.
- Distribute messages received to the supervised vertices.
- Start and end supersteps by the signal from the master.
- Signal communicator to receive messages from all vertices.
- Signal to start workers.

4.3.3 Architecture of Worker

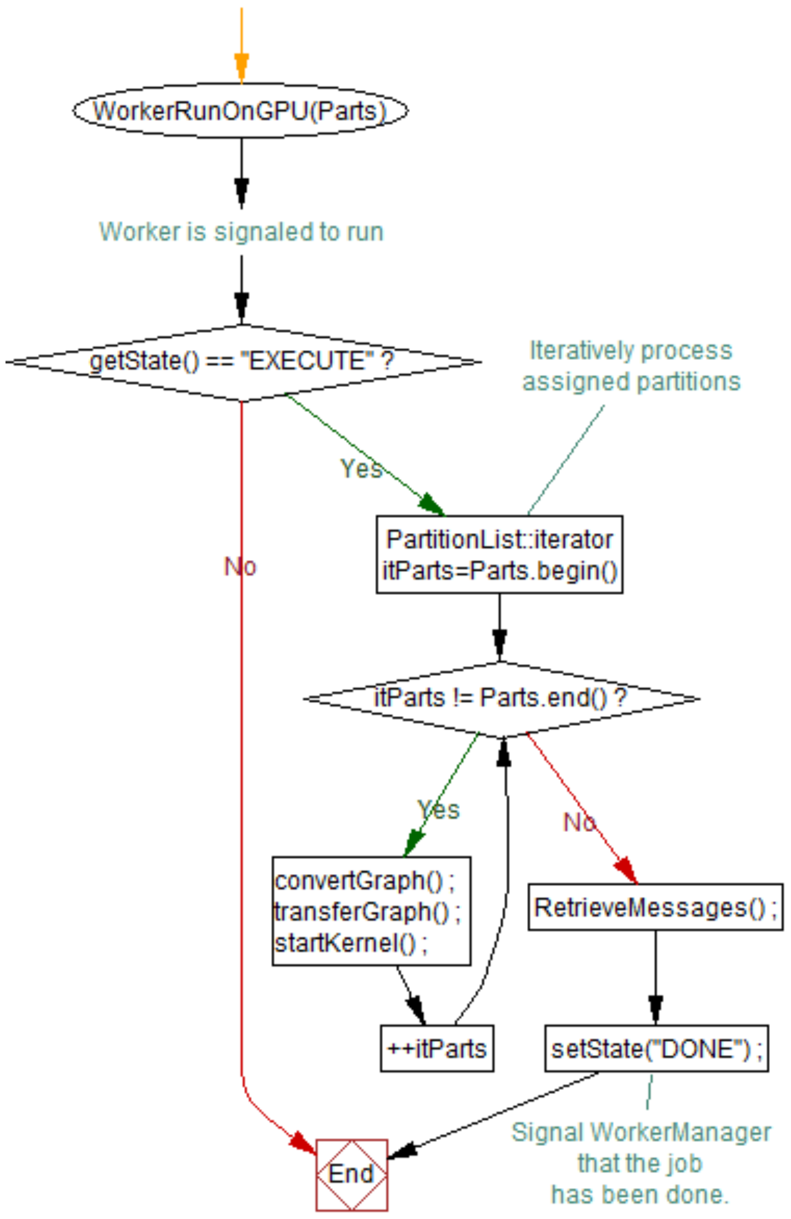

WorkerRunOnGPU(Parts)
Worker is signaled to run
getState() == "EXECUTE" ?
Iteratively process assigned partitions
Yes
No
PartitionList::iterator itParts=Parts.begin()
itParts != Parts.end() ?
Yes
No
convertGraph() ; transferGraph() ; startKernel() ;
RetrieveMessages() ;
++itParts
setState("DONE") ;
Signal WorkerManager that the job has been done.
End

Figure 4.9: Worker Architecture

A Worker corresponds to the processing of one or several partitions on the GPU. As is illustrated in Figure 4.9, the worker thread is initialized by the worker manager and waits for the signal to execute. As soon as it receives the “EXECUTE” signal, the worker converts the subgraph into the graph format that can be processed on the GPU. Then the worker iteratively transfers the graph onto GPU global memory. And the worker computes the execution thread dimensions before launching a batch of GPU threads to process the graph. Finally, messages are retrieved from GPU memory to the main memory and cached in the communicator.

The main functions of the worker are listed as follows:

- Convert graphs into the format that can be processed on the GPU.
- Transfer graphs to GPU global memory.
- Compute execution configuration and launch the kernel function.
- Retrieve messages from GPU global memory to the main memory.
- Signal worker manager that the job has been done.

4.3.4 Architecture of Vertex

Algorithm 7: Generic API For User Derived Vertex Class

```
class DerivedVertex : VertexInGPU {
public:
    __device__ void compute() {
        ...  // Algorithm implemented by users.
    }
    ...  // Other public members can be added by users.
private:
    ...  // Private members can be added by users.
} ;
__device__ VertexRegisterInGPU<DerivedVertex> reg ;
```

A vertex corresponds to a GPU thread in SIGPS. The vertex class is the interface between users and SIGPS. Users derive an application vertex subclass and implement the overloaded “compute” function. Algorithm 8 is the structure of the **Vertex** class. All the public methods are open APIs that can be called by users in their derived Vertex classes. In other words, programmers can utilize the inherited methods to implement their algorithms in the overloaded “compute” functions. Algorithm 7 is the generic API that a user can use to derive Vertex class and implement their algorithms.

Algorithm 8: Vertex Class

```
class VertexInGPU : VertexAPIGPU {
public:
    __device__ void initialize(int, double, int, int, message*, int, int, edge*, int,
int, message*) ;
    __device__ void initCompute() ;
    __device__ virtual void compute() = 0 ;

    // Auxiliary Member Retrieval and Access Methods
        ...
    // Messages Received Retrieval and Access Methods
        ...
    // Edges Retrieval and Access Methods
        ...
    // Messages Sending Method
        ...
private:
    int vertexId ;
    double vertexValue ;
    int superstep ;
    message *msgPtr, *msgSndPtr ;
    int msgSize, edgeSize, vertexSize ;
    edge* edgePtr ;
} ;
```

VertexRegisterInGPU is a struct implemented on GPU memory. It is designed for Vertex auto-registration in GPU. As is shown in Algorithm 7, users declare a global device object in line 7, after implementing their derived vertex classes. The name of the derived vertex class is used in the angular brackets to instantiate the template. Algorithm 9 displays the mechanism to realize the vertex class auto-registration. “Map” is a device hash table implemented for the map(registration) function.

Algorithm 9: Derived Vertex Class Auto-registration

```
template<typename T>
__device__ VertexInGPU* createVertexInGPU() { return (VertexInGPU*)(new T);
}

struct VertexFactoryInGPU {
        typedef Map<Key, VertexInGPU*(*)()> map_type ;

        __device__ VertexInGPU * createInstance(char* name, int len) {
            Key key(name, len) ;
            return getMap()->find_in_map(key)() ;
        }
        __device__ map_type * getMap() {
            if(!map) { map = new map_type; }
            return map ;
        }

        map_type * map ;
};

template<typename T>
struct VertexRegisterInGPU : VertexFactoryInGPU {
        __device__ void init(char* name) {
            int size = strlen(name) ;
            Key key(name, size) ;
            getMap()->insert_map(key, &createVertexInGPU<T>) ;
        }
};
```

The main functions of the vertex are listed as follows:

- Acts as a base class for user-derived vertex class.
- Provide a lists of public methods as API functions.
- Provide a virtual method “compute” for user to implement algorithms.
- Realize the derived vertex class auto-registration.

4.3.5 Architecture of Communicator

A communicator is a member object of the worker manager class. It handles all the messages within and among worker managers. Workers complete a superstep and send messages to the communicator. The communicator caches these messages and computes the destination for each one. Figure 4.10 illustrates the mechanism of a communicator under the worker manager. When the worker manager signals it to run, the communicator waits for the completion of all workers. While the function "communicate()" is invoked, the communicator clears the spooler queues for worker managers. Then it adds messages to the corresponding spooler queue according to the hostname(worker manager id), partition id, and destination vertex id. After sending all the messages to the spooler queues of different worker managers, the communicator signal its worker manager to stop the current superstep. Finally, the communicator flag itself to be in "STOP" state.

The main functions of the communicator are listed as follows:

- Poll until the worker manager signal to run.
- Wait for the completion of all the workers.
- Clear the spooler queues for all worker managers.
- Populate messages to its corresponding spooler queues.
- Send messages to its corresponding worker managers.
- Signal to stop workers.
- Signal to end superstep.
- Flag itself to "STOP".

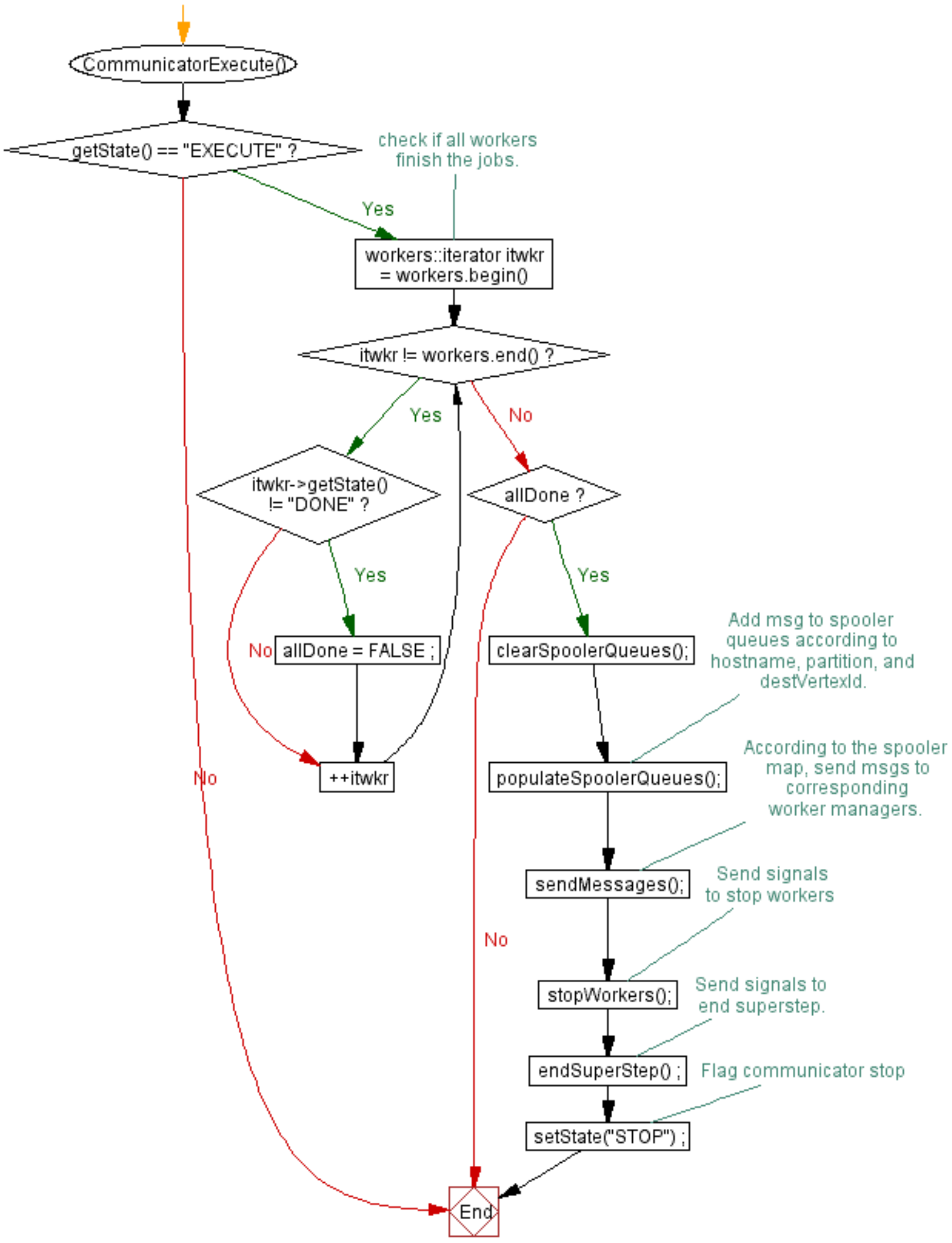

CommunicatorExecute()
getState() == "EXECUTE" ?
check if all workers finish the jobs.
Yes
workers::iterator itwkr = workers.begin()
itwkr != workers.end() ?
Yes
No
itwkr->getState() != "DONE" ?
allDone ?
Yes
Yes
No
allDone = FALSE ;
clearSpoolerQueues();
Add msg to spooler queues according to hostname, partition, and destVertexId.
No
++itwkr
populateSpoolerQueues();
According to the spooler map, send msgs to corresponding worker managers.
sendMessages();
Send signals to stop workers
No
stopWorkers();
Send signals to end superstep.
endSuperStep() ;
Flag communicator stop
setState("STOP") ;
End

Figure 4.10: Communicator Architecture

4.4 System Auxiliary Components

SIGPS system is implemented across CPU and GPU heterogeneous platform, using C++ and CUDA programming language. As a graph processing system, SIGPS consists of three main modules, namely “Graph”, “System” and “User API”. Each of these modules contains several functional classes. First of all, the “Graph” module consists of four classes, namely “Graph”, “Edge”, “Graph Partitioner” and “Graph Generator”. These classes realize a graph system that provides basic graph processing functions such as graph generation, graph reading, graph partitioning, critical value computation and result writing. Secondly, the “System” module consists of six classes, namely “Master”, “Worker Manager”, “Worker”, “DataLocator”, “MessageCenter” and “Message”. This module handles advanced graph processing such as parallelization, communication and BSP model. Lastly, the “User API” module provides a vertex interface for users to implement their own vertex class, which encapsulates user-defined graph processing algorithms. We have walked through the main components in Section 4.3. In the remaining portion of this section, a concise introduction of system auxiliary classes is provided.

4.4.1 Graph Generator and Graph Partitioner

Since SIGPS is a vertex-centric model, we define the graph in a format of the adjacency list. Each line of the input file is the neighboring list of one vertex. Specifically, the graph generator produces a synthetic graph in the adjacency list format. And the graph partitioner divides this original graph into several partitions that can be held in the system and GPU memory. We have proposed several partitioning algorithms in Section 3.5 and conducted a comprehensive comparison when partitioning a large graph. In order to compare the vertex-centric method with the edge-centric one, we provide a conversion method between the adjacency list and the edge flow format of the graph. Partitioner can also read in both formats of graphs.

4.4.2 Vertex API, Edge and Graph

For simplicity, vertex API is designed as an abstract class. Users only need to subclass the Vertex and override the compute member function. More specifically, in order to facilitate implementing user's algorithms in the simplest format, and in consideration of the C++ characteristic, we define the compute member function as an external function. When subclassing a Vertex class, users only need to structure their algorithms and implement them using the compute function template. Since the external compute functions are deployed to run on GPU, users need to send data parameters to the dedicated compiler as well. Besides the function interface, users can also design the output by themselves. The output data includes data sent back to standard output, data return by parameters and function return value. The object/struct/array data need to be returned by pointers for a better compatibility. Edge and Graph classes are fundamental for graph processing, which are instantiated when graph is generated. Besides basic operations, getting and setting the *SuperStep* are the most significant behaviors that a graph object operates under its processing model. In addition, basic graph algorithms can be implemented as well such as simple graph traversal and graph elements accessing.

4.4.3 Message Center and Data Locator

Communication is the principal system functionality that distinguishes SIGPS from Google's Pregel implementation. As mentioned above, Message Center receives messages sent by workers, does the shuffling and stores the associated data to the destination addresses of the messages. During each *SuperStep*, Data Locator calculates the message addresses in GPU memory for each worker. There are two types of shuffling methodologies implemented in Data Locator, data shuffling and thread shuffling.

Definition 4.4.1. *Collective Data Shuffling:*

Let X_i be the value of thread i before shuffling, Y_i be the index of the value after shuffling. Then Y_i satisfies $X_{Y_i} \leq X_{Y_j}$ for $i \leq j$.

Definition 4.4.2. *Comparison-based Collective Data Shuffling:*

Comparison-based Collective Data Shuffling enables users to define their own comparison function for data shuffling. Complicated algorithms can be applied in this case so that data shuffling can be controlled by the Data Locator.

Definition 4.4.3. *Collective Thread Shuffling:*

Let X_i be the value of thread i before shuffling, Z_i be the value of thread i after shuffling. Then after shuffling, threads are adjusted so that $Z_i \leq Z_j$ for $i \leq j$.

Definition 4.4.4. *Comparison-based Collective Thread Shuffling:*

Comparison-based Collective Thread Shuffling enables users to define their own comparison function for thread shuffling. Advanced shuffling strategies can be employed by the Data Locator so that data can be sent to corresponding recipients.

Example 5.

Threads with rank $\{0,\ 1,\ 2,\ 3,\ 4,\ 5\}$ compute values $\{X_0,\ X_1,\ X_2,\ X_3,\ X_4,\ X_5\} = \{8,\ 4,\ 2,\ 5,\ 1,\ 3\}$. After data shuffling, the returned index set $\{Y_0,\ Y_1,\ Y_2,\ Y_3,\ Y_4,\ Y_5\} = \{4,\ 2,\ 5,\ 1,\ 3,\ 0\}$. After thread shuffling, the value set $\{Z_0,\ Z_1,\ Z_2,\ Z_3,\ Z_4,\ Z_5\} = \{1,\ 2,\ 3,\ 4,\ 5,\ 8\}$.

After shuffling in general, message center stores the values according to the addresses computed by Data Locator. Actually, graph processing algorithms such as SSSP, PageRank, and Dense Subgraph Mining can be abstracted into several iterative *SuperSteps*, which can be further broken into a sequence of parallel constructs. So that data movement can be separated into a sequence of shufflings in the device memory.

The underlying system implementation of the shuffling has to realize data moving in the GPU memory hierarchy. There are two types of the data moving units, namely Move(<b_x, t_i>,<b_y, t_j>), Move(<b_x, t_i>,<b_x, t_j>). In addition, Data Locator also handles global memory coalescing, L1/L2 cache coherence and shared memory bank conflict issues.

Definition 4.4.5. *Data Moving Unit: Move(source, destination)*

Assume the function Move(source, destination) is the unit operation that Message Center executes to pass data from the address space of one thread to that of another. Let "source" and "destination" be the threads that mean to communicate, which are in the pair<blockId,threadId> format.

Definition 4.4.6. *Move(<b_x, t_i>,<b_x, t_j>): Intra-block Data Moving*

Communication occurs between two threads t_i and t_j from the same block b_x. Since there are caches [2] *and shared memory in the same block, most data moving within the same block can be operated in the block-wise shared memory. A shared memory read/write access pattern is exploited in order to avoid some potential bank conflicts. If the message is too large in size, a data replacement is needed to swap data via cache from global memory.*

Definition 4.4.7. *Move(<b_x, t_i>,<b_y, t_j>): Inter-block Data Moving*

Communication occurs between two threads t_i and t_j from different blocks b_x and b_y respectively. A global memory read/write access pattern is exploited for memory coalescing. Since shared memory is considered to be user manageable cache, the cache coherence issue is also important if the global memory has been updated.

[2] for Graphics Card with Compute Capability 2.x and 3.0

When a batch of threads are shuffled by Worker Managers, Move(<b_x, t_i>,<b_x, t_j>) and Move(<b_x, t_i>,<b_y, t_j>) unit operations are grouped separately by Message Center. By conducting a bundle of intra-block and inter-block data moving, Data Locator enhances the data transfer bandwidth, minimize communication overhead and accelerates the data moving process.

4.4.4 State Logging

For distributed system, state logging provides a solution for system fault tolerance. Message sent among remote peers might be lost during communication. Workers in the peer system might suffer failure as well. Whether a system can provide measures to recover the computation if some system failure or message lost was detected is vital for such system. However, for shared memory system, fault tolerance is not so important as a system design issue. Message passing is replaced by data moving in the emulating shared memory and worker threads provided by GPU seem to be stable during execution.

Logging the state of each *SuperStep* makes SIGPS adapted to various system environment. The state of one *SuperStep* consists of the state of all partitions, vertex values, edge values and messages received. Because SIGPS is based on BSP model, the states of all *SuperSteps* are recorded as a snapshot of the system. We have illustrated the block state machine in section 4.2. Here we rehearse it from the perspective of "logging". At the beginning of each *SuperStep*, block state is activated by Worker Manager. Then the block state may remain active until a block synchronization is executed, which marks an end to the current *SuperStep*. The state of a single partition is only one part of the snapshot of the whole graph. For each *SuperStep*, all partitions are processed separately. While the system has excessive resources, the whole graph can be processed concurrently. However, due to the possible limitation of the system resources, graph partitions may not be processed simultaneously. With limited resources provided, some partitions need to be swapped out so that other partitions can be processed. State logging is critical for this limited resources situation. An operation of the state logging before barrier synchronization stores the state for each partition. At the beginning of the next *SuperStep*, the state of the partition is restored before receiving messages from other partitions.

State logging for SIGPS is also designed to enhance the system efficiency, including the implementation and execution of the algorithms. By recording the state of one *SuperStep*, some algorithms can be speeded up since there is no need for redundant recomputation in every iteration. For example, some aggregator values do not need to be re-computed for every *SuperStep*. Additionally, logging the states of each *SuperStep* makes it easy for users or system programmers to debug their programs and have a detailed understanding of their algorithms.

For state logging implementation, we utilize boost file system library and serialization facilities to flatten the objects of SIGPS modules. Algorithm 10 is the code the serialization library invokes to save or load a class instance to/from an archive.

Algorithm 10: Object Serialization

```
friend std::ostream & operator<<(std::ostream &os, const class_name &or);
friend class boost::serialization::access;
template<class Archive, class T>;
inline void serialize( Archive & ar, T & t, const unsigned int file_version ) {
        // invoke member function for class T
        t.serialize(ar, file_version);
        // save/load class member variables
        ar & member ;
        // invoke serialization of the base class
        ar & boost::serialization::base_object<base_class_of_T>(*this);
}
```

4.5 Automatic Execution Configuration and Dynamic Thread Allocation

For graph processing on GPU, each vertex is represented by one GPU thread. Local computation in Figure 4.2 is conducted by the graphics processors, and message communication is emulated in the graphics memory. In Section 4.3, we have mentioned that SIGPS uses CUDA driver API to implement the underlying system functions. Therefore, SIGPS inherits CUDA thread hierarchy, which consists of grid, block and GPU light-weight threads. Grid corresponds to the execution of a device kernel function. Block refers to a batch of GPU light-weight threads that can communicate via on-chip fast caches and synchronize by multi-processor barriers with each other. A grid consists of a number of blocks that can be assigned to saturate available GPU multi-processors concurrently. As soon as a block of threads finish their task on one multi-processor, the system will assign a new block of waiting threads to run on that multi-processor.

Instead of waiting for users to decide the execution configuration parameters, SIGPS defines the properties of the thread model, which simplifies the programming and hides the system complexity. Table 4.1 lists the pre-set properties of SIGPS thread model. Since the system model is vertex-centric, each vertex is designed to be represented by one thread. According to the current GPU capabilities, a block can afford 512 threads in terms of the amortized shared resources. The number of blocks is then decided by the task amount and the unit size. It is worth mentioning that the number of threads which are actually running in parallel is decided by the multiprocessor size, the amount of shared resources such as registers and the thread model configuration. All the other dimensions of the grid and block are simply set to be 1. The whole thread model is shaped to be in one dimension and the corresponding memory space of all threads is packed into a linear array.

Property	**Value**	**Description**
thread.size	$Vertex.size$	GPU thread number to be launched.
thread.grid.size	ceil($\frac{Vertex.size}{512}$)	The x dimension of thread grids.
thread.block.size	512	The x dimension of thread blocks.
thread.grid.sizey	1	The y dimension of thread grids.
thread.block.sizey	1	The y dimension of thread blocks.
thread.block.sizez	1	The z dimension of thread blocks.

Table 4.1: GPU Thread Configuration

Compared with other graph processing systems, SIGPS is equipped with new features that manage to enhance the processing efficiency. Firstly, besides the default thread configuration, in SIGPS users can allocate more threads as is in need. There are two levels of parallelism in graph processing. One is the parallelism among vertices of the graph. The other one is the task parallelism inherent in graph algorithms. The dynamic thread allocation enables users to exploit a finer parallelism for the problem. The *forall* construct can be unfolded to explore an additional in-task parallelism, such as a vertex inquiries around all its neighbors to get the associated messages. Moreover, the in-block and intra-block parallelism is explored by Worker Manager and Worker hierarchical structures to distribute the tasks. Secondly, SIGPS employs an efficient resizable list in the implementation of Message Center. As we mentioned above, Message Center manages the aggregated memory space for all threads in GPU memory. Due to the simplified one dimension thread model, Message Center is also shaped to be like a linear array. The capability of resizable list allocation makes SIGPS efficient and effective for dynamic threads. Specifically, when a batch of threads are dynamically launched, SIGPS resizes the linear array of Message Center. Thread ranks are also re-assigned to match the linear formation of the memory space. Thirdly, SIGPS simplifies the communication by simply getting and putting values to their memory spaces. Fourthly, SIGPS stores graph class members separately in aligned arrays of basic types. Memory access of graph data is always coalesced if only all members satisfy coalescing requirements. Last but not least, Worker Manager and Worker are implemented in a sequential programming style, successfully avoiding the complexity of parallel programming and leaving complicated parallelization and optimization to the compiler.

4.6 Case Study

4.6.1 Case One: PageRank

In web analysis field, PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. It is represented by a link analysis algorithm, which can be applied to any collection of entities with reciprocal quotations and reference. Specifically, a numerical weighting is assigned to each element of hyper-linked set of documents in order to “measure” its relative importance within the set. The computations of the algorithm have several passes, called “iterations” through the collection to adjust approximate PageRank values to more closely reflect the theoretically true value. Generally in mathematics, the PageRank value for any page **u** can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)\,[3]} \approx 0.15 \times \frac{1}{|V|} + 0.85 \times \sum \frac{PR(v)}{L(v)}$$

To implement the PageRank algorithm using SIGPS, we extend the Vertex class on the GPU to create a **PageRankVertex** user-defined GPU class. The PageRankVertex class derives the **compute()** virtual method, executed by one GPU thread, to calculate the PageRank value of the vertex.

The initial PageRank value of each vertex is set to be $\frac{1}{|V|}$[4]. The process of PageRank computations consists of several **superstep**s. Within each superstep, vertex thread receives messages from its neighbors, computes its tentative PageRank and sends messages to its neighborhood along its outgoing edges.

The algorithm converges when the PageRank values of all vertices become stable. In reality, we employ a “Pay as you go” strategy and pre-set the number of supersteps to simplify the computation.

Algorithm 11 is an example of user-defined PageRankVertex class implemented for GPU execution. We take advantage of the generic Vertex APIs to derive the function “compute()”. In line 7 to 9, the GPU thread reads messages from the neighborhood of the vertex. Line 10 computes the tentative PageRank value for the current superstep. In line 16 to 18, the vertex issues its tentative PageRank divided by the number of links as messages to its neighboring vertices. After implementing the PageRankVertex class, users need to declare a global variable, instantiated by the class name **“PageRankVertex”** in line 23, to register this customized vertex class in the system. The “reg” object will be utilized to initialize the PageRankVertex class and create vertices objects in the system automatically.

Algorithm 11: PageRankVertex

```
1  class PageRankVertex : VertexInGPU {
2  public:
3      __device__ void compute() {
4          if ( this->getSuperStep() >= 1 ) {
5              double sum = 0 ;
6              int numMessages = this->getMsgSize() ;
7              for ( int i = 0 ; i < numMessages ; ++i ) {
8                  sum += this->getMsgValues(i);
9              }
10             double newPageRank = 0.15 / this->getTotalNumVertices() +
   0.85*sum ;
11             this->setValue(newPageRank) ;
12         }
13
14         if ( this->getSuperStep() < SUPERSTEP_NUM ) {
15             int numEdges = this->getEdgeSize() ;
16             for ( int i = 0; i < numEdges; ++i ) {
17                 this->sendMessage(this->getEdge(i), this->getValue()/numEdges) ;
18             }
19         }
20     }
21 };
22
23 __device__ VertexRegisterInGPU<PageRankVertex> reg ;
```

4.6.2 Case Two: Single Source Shortest Path

In graph theory, the Single Source Shortest Path problem is the problem of finding a path between vertices in a graph so that the sum of the weights of the edges in the path is minimized. In mathematics, let $G = (V, E, C)$ be a directed/undirected graph and let s be a designated vertex in V. Compute $Min(s, x; V)$ for every vertex $x \in V$. There are several classic algorithms to solve this problem, such as the Dijkstra's algorithm and the Bellman/Ford algorithm.

In order to solve Single Source Shortest Path problem, we implement the Dijkstra's algorithm using SIGPS. We derive the GPU vertex class and define an "SSSPVertex" and its member function "Compute()". During execution, each GPU thread create an "SSSPVertex" object and run the function "Compute()" to calculate the shortest distance.

The algorithm converges when all the vertex values are set to be the shortest distances to the source vertex. There will be several passes before all vertices vote to stop, and the system forces all vertex threads to synchronize with each other between subsequent supersteps.

Algorithm 12 is an example of user-defined SSSPVertex class implemented for GPU execution. We take advantage of the generic Vertex APIs to derive the function "Compute()". Initially, we set the pre-vertex-id of the current vertex to be "-1" in line 4. We also utilize a temporary shortest distance and the preNode to record the tentative information. From line 7 to 12, the shortest distance is initialized for each vertex. The vertex threads read messages from their neighbors in the "for-loop" from line 14 to 18. If the tentative computed shortest distance is smaller than the vertex value, the vertex value will be replaced by this smaller distance and its pre-vertex-id information will be updated as well. Finally, the vertex thread sends its updated distance added by the corresponding edge cost as a new message to all its neighbors.

Algorithm 12: SSSPVertex

```
class SSSPVertex : VertexInGPU {
public:
    __device__ void compute() {
        this->preVertexId = -1 ;
        int preNode = -1 ;

        double distance ;
        if ( this->getVertexID() == 0 ) {
            distance = 0 ;
        } else {
            distance = INFINITY ;
        }
        int numMessages = this->getMsgSize() ;
        for ( int i=0 ; i<numMessages ; ++i ) {
            if ( this->getMsgValues(i) < distance ) {
                distance = this->getMsgValues(i) ;
                preNode = this->getMsgSourceVid(i) ;
            }
        }
        if ( distance < this->getValue() ) {
            this->preVertexId = preNode ;
            this->setValue(distance) ;
            int numEdges = this->getEdgeSize() ;
            for ( int i=0; i<numEdges; ++i ) {
                this->sendMessage(this->getEdge(i), distance +
this->getEdgeCost(i)) ;
            }
        }
    }

private:
    int preVertexId ;
};

__device__ VertexRegisterInGPU<SSSPVertex> reg ;
```

4.6.3 Case Three: Dense Subgraph Mining

To parallelize dense subgraph mining algorithm using SIGPS, we implement DN-graph mining algorithm by extending the Vertex class on GPU to create a “DSMVertex” user-defined class. The “DSMVertex” class derives the **compute()** virtual method, executed by one GPU thread, to calculate the $\tilde{\lambda}$ value of the vertex.

The algorithm converges when the “converge” flag is true, which means all vertices have updated their $\tilde{\lambda}$ value in the current superstep. Algorithm 13 is an example of user-defined “DSMVertex” class implemented for GPU execution. We utilize the generic Vertex APIs to describe the behavior of each vertex. At first, we pre-set the “converge” flag to be TRUE. The “for-loop” from line 6 to 20 queries all the neighbors of the current vertex that had sent him a message in the last superstep. Line 7 computes the common neighbors of the vertex and its neighbor and the estimated $\tilde{\lambda}$ value is bounded in line 11. If this estimated $\tilde{\lambda}$ value is bigger than the current value of the vertex, it is said to be supported by this neighbor and the counter “support” is added in line 13. If the supporting vertices are found smaller than the $\tilde{\lambda}$ value of the current vertex, it will be updated as the new $\tilde{\lambda}$ value of the vertex. And the flag “converge” will be set to be false. This indicates that the new value needs to be spread to its neighbors, which is sent out as messages in the loop (line 22 to 24).

Algorithm 13: Dense Subgraph Mining Compute Function

```
class DSMVertex : VertexInGPU {
public:
    __device__ void compute() {
        converge = TRUE ;
        int numMessages = this->getMsgSize() ;
        for ( int m=0 ; m<numMessages ; ++m ) {
            int coNeighborSize = neighborhood_join (this->getNeighbors(),
m.getDestVertex().getNeighbors(), coNeighbors) ;
            int support = 0, *coNeighbors, density = coNeighborSize ;
            for ( int j=0 ; j<coNeighborSize ; ++j ) {
                coNeighbor = coNeighbors[j] ;
                density = Min(Min(this->getNeighborValue(coNeighbor),
this->getMsgValue(m)), Min(this->getNeighborValue(coNeighbor),
this->getValue()));
                if ( density ≥ this->getValue() ) {
                    support++ ;
                }
            }
            if (support < Min(this->getValue(), this->getMsgValue(m)) {
                this->setValue(this->getValue()-1);
                converge = FALSE ;
            }
        }
        if (!$converge$) {
            for ( int e=0 ; e<this->getEdges() ; ++e ) {
                this->sendMessage(this->getEdge(e), this->getValue()) ;
            }
        }
    }
};

__device__ VertexRegisterInGPU<DSMVertex> reg ;
```

4.7 Generic Vertex APIs Usage

In this section, we provide an elementary tutorial about how to use the vertex APIs provided by SIGPS. In Section 4.3.4, we have introduced the basic structure of the generic vertex APIs and the Vertex class. In this section, we re-list Algorithm 7 as Algorithm 14. Users derive the Vertex class in GPU as displayed in Algorithm 14. They can add public/private member variables and methods they need. Just remember to add "__device__" in front of the methods since they are required to work on GPUs.

Algorithm 14: Generic API For User Derived Vertex Class

```
class DerivedVertex : VertexInGPU {
public:
    __device__ void compute() {
        ...   // Algorithm implemented by users.
    }
    ...   // Other public members can be added by users.
private:
    ...   // Private members can be added by users.
} ;
__device__ VertexRegisterInGPU<DerivedVertex> reg ;
```

We list the full declaration of the Vertex class in Algorithm 15. The **initialize(...)** and **initCompute()** methods are used by the system to initialize the vertex and invoke the "compute()" method. They are functions that connect the system to the code implemented by users.

The virtual method **compute()** is the core function that must be implemented by users in order to run their own algorithms to do graph processing.

The methods **getVertexID()** and **setVertexID(int)** are functions to retrieve and access the vertex id that can be used to identify the vertex in the graph.

The methods **getTotalNumVertices()** and **setTotalNumVertices(int)** are functions to retrieve and access the size of the graph (total vertex number).

The methods **getValue()** and **setValue(double)** are functions to retrieve and access the value attached to the vertex which can be utilized by users. The meaning of the vertex value can be defined by users in their algorithms. For example, in "*PageRankVertex*" example, the vertex value is defined as the PageRank value of the current page node. And in "*SSSPVertex*" example, the vertex value is defined as the shortest distance to the source from the current vertex.

The methods **getSuperStep()** and **setSuperStep(int)** are functions to retrieve and access the tentative superstep.

The method **getMessages()** is the function to get the starting positions of the received messages list. The method **getMsgSize()** is the function to get the size of the received messages list. With the above two methods, users can retrieve and access all messages received by the vertex thread.

The methods **getMsgValue(int)**, **getMsgSourceVid(int)**, **getMsgDestVid(int)** and **getMsgSuperStep(int)** are functions to retrieve the components of the received messages according to the index (0 . . . getMsgSize()) in the messages list. Using these methods, users can retrieve the value sent accompanied with the message. In each superstep, messages are issued with the source/destination vertex ids of the message.

The method **getEdges()** is the function to get the starting positions of the contingent edges list. The method **getEdgeSize()** is the function to get the size of the contingent edges list. With the above two methods, users can retrieve and access all the contingent edges of the current vertex.

The methods **getEdgeCost(int)**, **getEdgeSourceVid(int)** and **getMsgDestVid(int)** are functions to retrieve the components of an edge according to the index (0 . . . getEdgeSize())

in the contingent edges list. Using these methods, users can retrieve the edge cost, and the source/destination vertex ids of the edge.

The method **sendMessage(edge e, double value)** is the function for the vertex thread to send its value along an edge e, encapsulated as a message.

Algorithm 15: Vertex Class

```
class VertexInGPU : VertexAPIGPU {
public:
    __device__ void initialize(int, double, int, int, message*, int, int, edge*, int,
int, message*) ;
    __device__ void initCompute() ;
    __device__ virtual void compute() = 0 ;

    __device__ int getVertexID() ;
    __device__ void setVertexID(int) ;
    __device__ int getTotalNumVertices() ;
    __device__ void setTotalNumVertices(int) ;
    __device__ double getValue() ;
    __device__ void setValue(double) ;
    __device__ int getSuperStep() ;
    __device__ void setSuperStep(int) ;

    __device__ message* getMessages() ;
    __device__ double getMsgValues(int) ;
    __device__ int getMsgDestVid(int) ;
    __device__ int getMsgSourceVid(int) ;
    __device__ int getMsgSuperStep(int) ;
    __device__ int getMsgSize() ;

    __device__ edge* getEdges() ;
    __device__ edge& getEdge(int) ;
    __device__ int getEdgeSourceVid(int) ;
    __device__ int getEdgeDestVid(int) ;
    __device__ double getEdgeCost(int) ;
    __device__ int getEdgeSize() ;

    __device__ void sendMessage(edge, double) ;

private:
    int vertexId ;
    double vertexValue ;
    int superstep ;
    message *msgPtr, *msgSndPtr ;
    int msgSize, edgeSize, vertexSize ;
    edge* edgePtr ;
} ;
```

4.8 Experiments

In this section, we study the performance of SIGPS system. We implement three algorithms, namely PageRank, SSSP (Single Source Shortest Path) and DSM (Dense Subgraph Mining) using the generic APIs of the system.

4.8.1 Experimental Settings

We have conducted our main experiments on three low-end GPU accelerated desktop systems. They are a DELL PC equipped with a low-end NVIDIA GeForce 9400 GT graphics card, and two ACPI x64-based systems accompanied with an NVIDIA GeForce GT 520 graphics card and an NVIDIA GeForce GT 330 graphics card respectively. The price of the GeForce 9400 GT and GeForce GT 520 graphics cards are both less than 50 US dollars. And a GeForce GT 330 graphics card is no more than 100 US dollars. The whole GPU-accelerated personal computer systems cost only around 1000 US dollars, which are quite affordable for ordinary users.

In order to study the computational capabilities of the SIGPS system, we also adopted an exhaustive study of the desktop system resources. In this study, we generate a series of directed graphs with ten thousands to one million vertices on a desktop equipped with an NVIDIA GeForce GTX 760 graphics card. The PageRank algorithm is executed on the massive graphs to test the computing scalability of the SIGPS system. We present the major technical specifications of our four platforms in Table 4.2.

We employ synthetic and real datasets in our experiments. Experimental synthetic graphs are generated by the system graph generator component. We use the graph density, which is defined to be $D = \frac{2|E|}{|V|(|V|-1)}$. For vertex centric algorithms, a series of graphs

Platform	**CPU**	**Main Memory**	**GPU**	**GPU Memory**	**Compute Capability**	**GPU Price**
DELL Opti-755 Geforce 9400 GT	2 cores 2.33 GHz	3.3 GB DDR2	1×16 cores 550 MHz	512 MB 400 MHz	1.1	< \$50
ACPI x64-based Geforce GT 520	4 cores 3.40 GHz	8 GB DDR2	1×48 cores 1.62 GHz	993 MB 535 MHz	2.1	< \$50
ACPI x64-based Geforce GT 330	4 cores 3.40 GHz	8 GB DDR2	12×8 cores 1.34 GHz	1 GB 790 MHz	1.2	< \$100
i7-4770 x64-based Geforce GTX 760	4 cores 3.40 GHz	16 GB DDR2	6×192 cores 1.15 GHz	4 GB 3.004 GHz	3.0	< \$500

Table 4.2: Experimental Platforms

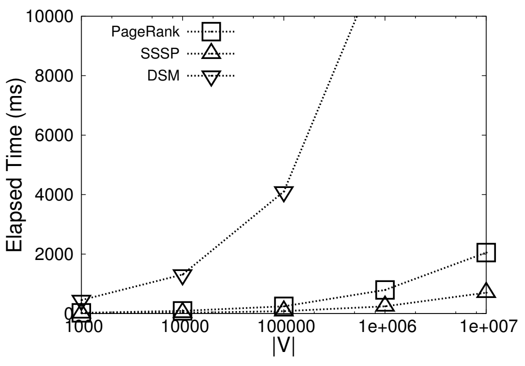
with varying vertex sizes from 10^3 to 10^7 are created. The edge sizes can be decided by $|E| = D\times(|V|(|V|-1))$. For edge centric algorithms, graphs with edges ranging from 10^4 to 10^8 are also produced. The corresponding vertex sizes are calculated by $|V| = \lceil\sqrt{\frac{2|E|}{D}}\rceil$. Real graphs in our experiments include flickr, DBLP, PPI, and Netflix datasets. Flickr graph is derived from a well known photo sharing social network. Each node in the graph represents one person and an edge between two persons denotes this two persons share photos with each other. There are 1,715,255 people and 22,613,982 sharing relationship recorded in the graph. Since interactions between proteins are important for the majority of biological functions, Protein Protein Interaction (PPI) graphs are useful tools to study the behavior of the entire interactomics system of any living cell. Our PPI graph contains 17203 interactions among 4930 proteins. Netflix is an American provider of on-demand internet streaming media. There are 480,000 customers and 17,000 movies in the dataset. Table 4.3 lists the graph data information in our study.

4.8.2 Scalability Study

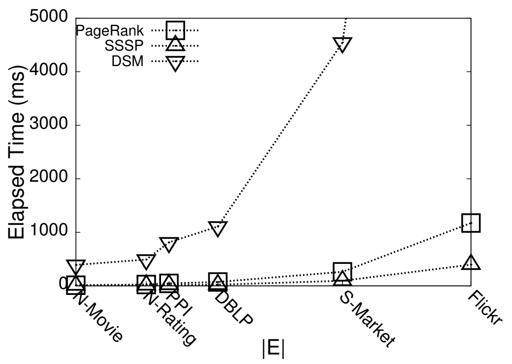
In order to study the scalability of our system, we run the three graph processing algorithms, PageRank, SSSP, and DSM with increasing graph sizes. Figure 4.11 illustrates the growing tendency of the running time of the corresponding algorithms. When graph

Graph Types	Graph Name	|V|	| E|
Synthetic	-	$[10^3 \sim 10^7]$	$[10^4 \sim 10^8]$
Real	Flickr	1715255	22613982
Real	DBLP	23136	54989
Real	PPI	4930	17203
Real	Netflix Movie	1000	1881
Real	Netflix Rating	1000	10037
Real	Stock Marcket	6018	1064138

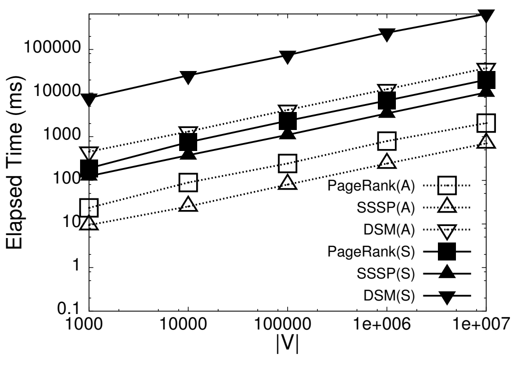
Table 4.3: Experimental Datasets



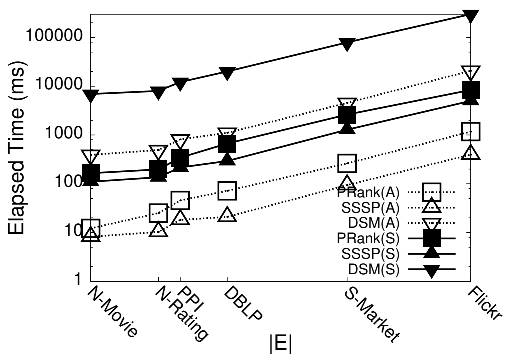
(a) Synthetic Graphs



(b) Real Graphs



(c) Synthetic Graphs



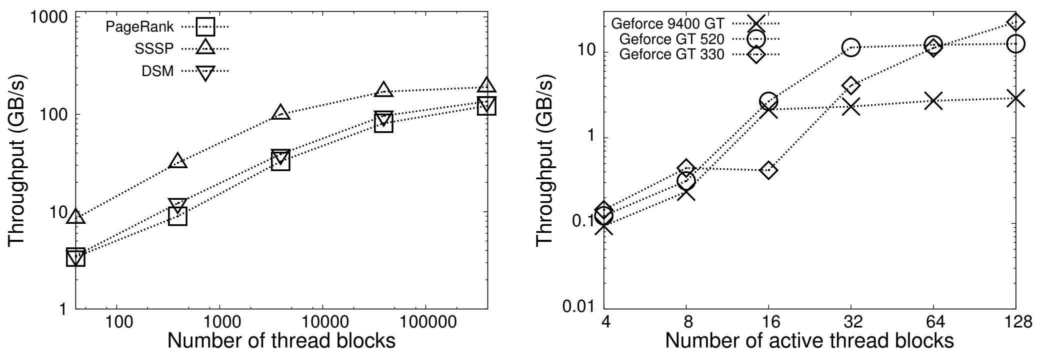
(d) Real Graphs

Figure 4.11: System Scalability

size increases exponentially, the elapsed time of processing a graph rises up with an accelerated speed. In Figure 4.11(c), we applied PageRank, SSSP and DSM algorithms on synthetic graphs with vertex size ranging from 10^3 to 10^7. We can observe that among all three processing algorithms, dense subgraph mining costs the most amount of time.

Figure 4.11(b) displays the execution time when the algorithms are run on real graphs. As graph size increases, dense subgraph mining algorithm uses more time. Even though there are more nodes in the DBLP graph, all three algorithms take longer time to run on the stock market graph. This is because the stock market graph is denser than the DBLP one. Moreover, the three algorithms are processed in both sequential mode (CPU only) and GPU accelerated mode of our system. Figure 4.11(c) compares the running time of both modes when the algorithms are run on the synthetic graphs. Both x-axis and y-axis are in log-scale. We can observe linear growth of the execution time when the algorithms are processed in both sequential and GPU-accelerated modes. Similarly, linear curves showing the algorithms running on the real graphs in Figure 4.11(d) also prove good scalability of the system. The axes in figures 4.11(b) and 4.11(d) are arranged according to the edge sizes of the real graphs.

4.8.3 Communication Study



(a) Scalable Thread Blocks (b) Active Blocks

Figure 4.12: Communication Throughput

To study the communication cost of the system, we log the data movement, calculate the communication throughput and plot them in Figure 4.13. Specifically, Figure 4.12(a) shows memory throughput from a data movement point of view, which is computed by

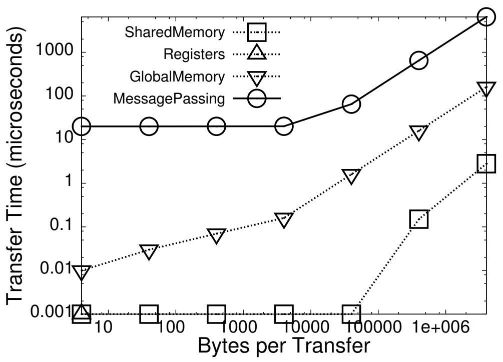


Figure 4.13: Communication Cost

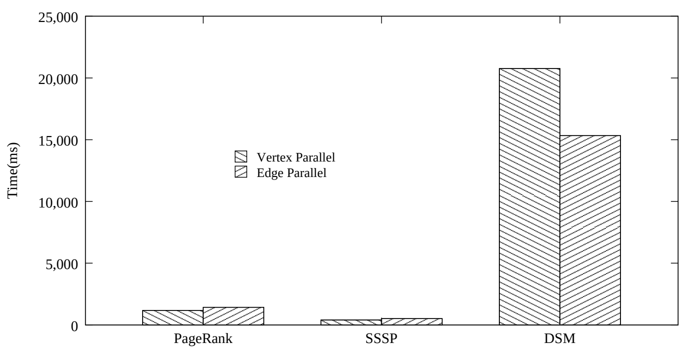


Figure 4.14: Vertex Parallel vs Edge Parallel

dividing the estimated sum of message sizes by GPU running time. While Figure 4.12(b) shows memory throughput from a data access point of view, which is computed by dividing the estimated sum of the sizes of the messages accessed by GPU processors by GPU memory access time. Figure 4.12(a) displays the linear increase in the communication throughput with the increasing number of thread blocks, which is decided by the graph size. Among the three algorithms, PageRank has the lowest throughput since it produces the smallest amount of messages and consumes more time than SSSP. The throughput curves are slowing down as the graph becomes large enough. Figure 4.12(b) illustrates the communication throughput with a growth in the number of active thread blocks for different graphics cards. We can observe a drop at around 12 active thread blocks for Geforce GT 330 graphics card, which suggests that the multi-processors are being saturated by the active thread blocks. Additional data moving requests are being absorbed by fast caches. Finally, it can have a higher throughput since it has more processing resources.

To further study the data moving cost, we increase message size and compare data moving in different memory types. Figure 4.13 displays the data transfer time when the message size is increased. There is only one point for registers in the plot, because registers are used only for basic data types. Shared memory can be used as fast 64KB independent caches, which takes only 1-2 clock cycles for one access. L1/L2 caches and coalescing global memory access make global memory fast before data transferred become too large. In comparison, we emulate message passing by sending data to the system main memory and sending it back onto global memory. We can observe that message passing model is by no means much slower than our communication model.

4.8.4 Vertex Parallel vs Edge Parallel

In this experiment, we study the vertex centric methods. Our GPU-accelerated graph processing model maps graph nodes to GPU threads. We call this the *vertex-centric* (or *vertex-parallel*) approach. In vertex centric algorithms, the inner *forall*-loop over the message list of the vertex v iterates the node neighbors $N(v)$ and reads the messages sent from the neighbors. The algorithm also iterate across edges incident to the vertex before sending out its own value. More threads are dynamically launched to represent the neighbor nodes of the current vertex in these loops. If we instead parallelize over the edges directly, then each thread can represent one edge. We implement an alternative algorithm (Algorithm 16) to compare with the vertex centric model. Figure 4.14 shows the running speed comparison among three graph processing algorithms. SSSP and PageRank take a moment to finish while DSM runs much slower in both models. From the histogram, graph processing algorithms like SSSP and PageRank with simple logic run faster in vertex parallel model. Because each edge in the edge centric model needs to be processed in two directions of the data movement, the redundant processing and data update conflicts tradeoff the additional parallelism gained from the model. On the contrary, the edge parallel model simplifies the complicated processing logic of DSM, which speeds up the algorithm and outperforms its vertex parallel counterpart.

Algorithm 16: Edge-centric model

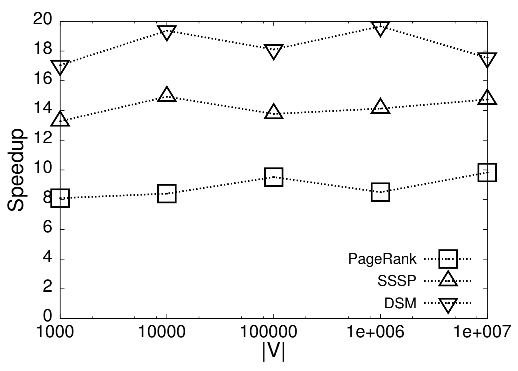
```
forall the Edge e=0:(Graph.edges.size-1) do
    e.compute() {
        u = e.getSource() ;
        v = e.getDest() ;
        value = Proc(e.Thread.get()) ;
        if u.isGood(value) then
            u.setValue(value) ;
        end
        value = u.getValue() ;
        if u.isGood(value) then
            e.Thread.put(value) ;
        end
    }
```

4.8.5 Speedup

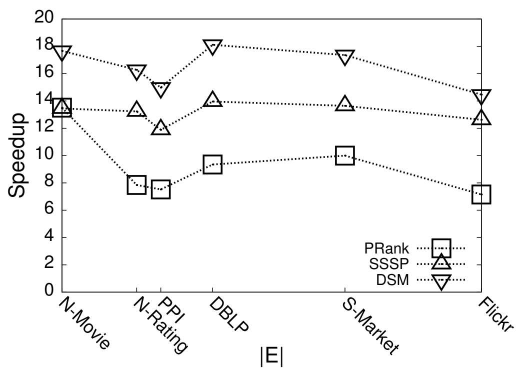
In this experiment, we study the speedup of our graph processing model. We compare all the three algorithms in parallel and sequential modes in Figure 4.15. Figure 4.15(a) illustrates the speedups when the algorithms run on synthetic graphs. From the plot, the speedup curves are steady for all three algorithms. Specifically, PageRank gains a speedup of about 8, while SSSP runs nearly 8 times faster and DSM gets a speedup as high as 16 to 17. Similarly, Figure 4.15(b) displays the speedups when the algorithms run on real graphs. PPI dataset and flickr graph have lower speedup compared with others. Both of the graphs have dense regions with a much higher local density than their neighboring areas. The imbalance of the tasks for threads forces low-degree vertices wait for their high-degree neighbors at the point of barrier synchronization, which slows down the system processing speed.

4.8.6 Comparable Experimental Study

In order to study the SIGPS system, we compared the three versions of the PageRank algorithm as follows.



(a) Synthetic Graphs



(b) Real Graphs

Figure 4.15: Speedup Study

First of all, we write the PageRank algorithm in a C routine that is only run by the desktop central processors, which is also named as “CPU routine of PageRank”. PageRank algorithm computes one vertex’s PageRank value using the PageRank value of all its neighbors. This leads to an iteration within the body of which the processor travels along the graph to update the PageRank values for all vertices. An initial PageRank value for all vertices are needed and an ending condition is set for a termination of the process. Figure 4.16 lists the main components of the “CPU routine of PageRank”. Specifically, Algorithm 17 calculates the sum of out degree for each vertex. Within the two embedded loops, CPU travels through the graph, pre-computes the out-degree of each vertex and saves the sum into the array “sumOfOutDegree”, which records the out-degree values of each vertex. Algorithm 18 travels the graph for the second time and computes the new PageRank value for each vertex using the sum of all the vertex’s neighboring PageRank value divided by the out-degree of the neighbor. Algorithm 19 displays how to compute the end condition.

Secondly, we implement the PageRank algorithm using pure CUDA programming model. Figure 4.17 lists the pure cuda counterparts of the “CPU routine of PageRank”, which we name it as “Pure CUDA Routine of PageRank”. Instead of using embedded

Algorithm 17: Calculate the sum of out degree of each vertex

```
forall the $i$ < numberOfVertex do
    sum = 0 ;
    forall the $j$ < numberOfVertex do
        sum + = *(Graph + i×numberOfVertex + j) ;
    sumOfOutDegree [i] = sum ;
```

Algorithm 18: Calculate the new PageRank value

```
forall the $i$ < numberOfVertex do
    sum = 0, k = 0, j = i ;
    forall the $j$ < numberOfVertex × numberOfVertex do
        if $*(Graph + j) == 1$ then
            if sumOfOutDegree $[k]! = 0$ then
                sum += PR[k] / sumOfOutDegree [k] ;
        k++, j += numberOfVertex;
    PR_Temp[i] = (1 - $\alpha$) + $\alpha$×(sum) ;
```

Algorithm 19: End Condition

```
bool END( a[], b[] ) {
    sum = 0 ;
    forall the i < numberOfVertex do
        sum += abs(a[i] - b[i]) ;
    if sum < END_WEIGHT then
        return true ;
    return false ;
}
```

Figure 4.16: CPU Routine of PageRank

Algorithm 20: Use CUDA to Calculate the sum of out degree

```
1 __global__ void calculateSumOfOutDegree(*sumOfOutDegree, *Graph) {
2     index = blockDim.x × blockIdx.x + threadIdx.x ;
3     if index < numberOfVertex {
4         sumOfOutDegree [index] = 0 ;
5         forall the $j$ < numberOfVertex do
6             sumOfOutDegree [index] += *(Graph +
      index×numberOfVertex + j) ;
7     }
8 }
```

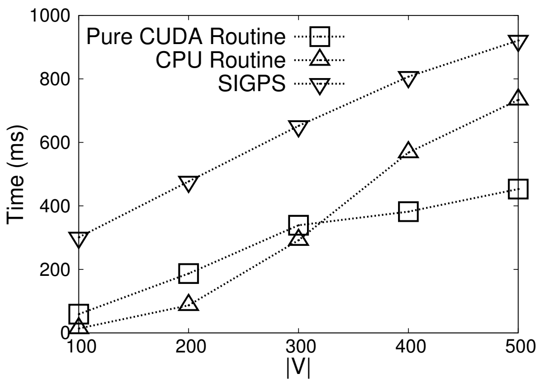
Algorithm 21: CUDA Kernel Function PRAdd: Calculate the PageRank value

```
1 __global__ void PRAdd(*PR, *Graph, *sumOfOutDegree) {
2     index = blockDim.x × blockIdx.x + threadIdx.x ;
3     if index < numberOfVertex {
4         sum = 0, k = 0, sumOfOutDegree [index] = 0, j = index ;
5         forall the $j$ < numberOfVertex × numberOfVertex do
6             if $*(Graph + j)$&&sumOfOutDegree$[k]$ {
7                 sum += PR[k] / sumOfOutDegree [k] ;
8             }
9             k++, j += numberOfVertex;
10        PR[index] = (1 - $\alpha$) + $\alpha$×(sum) ;
11    }
12 }
```

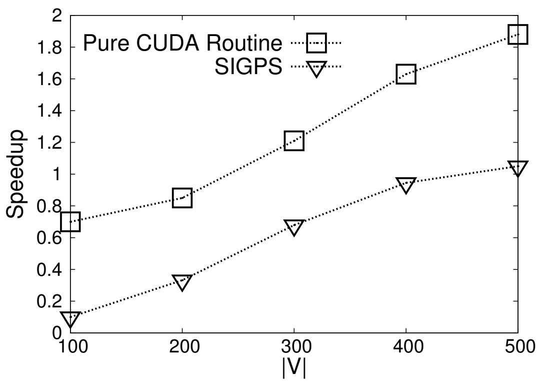
Figure 4.17: Pure CUDA Routine of PageRank

loops to travel through a graph by one CPU thread, Algorithm 20 applies CUDA programming model and employs a batch of parallel GPU threads to handle vertices. Each GPU thread executes one vertex and is indexed by the formula in the 2nd line. Therefore, for each GPU thread, there is only one iteration displayed from line 5 to 6, which is used to calculate the "sumOfOutDegree" for the vertex it handles. Algorithm 21 is a kernel function that is used to compute the PageRank on CUDA-enable GPUs. Similarly, there is only one iteration in the kernel, which reduces the complexity of the algorithm.

The third version is what we implemented for the SIGPS as stated in Algorithm 11. The vertex API simplified the algorithm and the system handles all tuning of the CUDA execution configuration.



(a) Execution Time



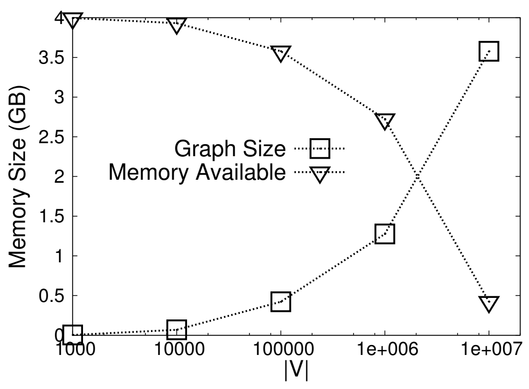
(b) Speedup

Figure 4.18: PageRank Methods Comparison

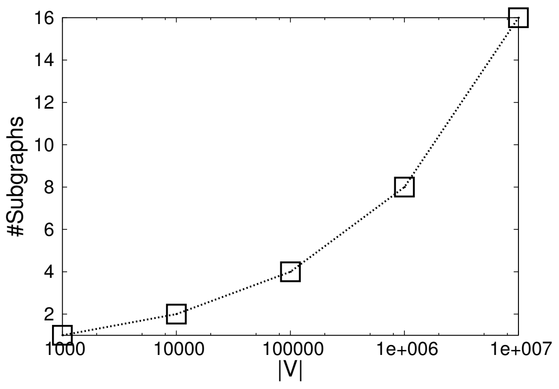
Figure 4.18 compares all the three versions of PageRank algorithm. Specifically, in Figure 4.18(a) all the three methods take more execution time while the graph size increases. One one hand, when the vertex size is less than 300, the CPU method is the fastest among the three routines. This is because the parallelism of the algorithm over small graph is low and GPU-accelerated routines have extra overheads. On the other hand, when the graph becomes large enough, the sufficient parallelism makes the CUDA routine run faster than the CPU one. Noticeably, SIGPS is the slowest among the three methods. This is caused by the systematic cost as well as the synchronization overhead. SIGPS has seemingly worse performance than CPU routine or pure CUDA one. However, the CPU and pure CUDA routines are stuck when the vertex size of the graph is larger than 500. These two routines can not handle the situation that a graph is too large to be stored in the memory. In contrast, SIGPS is equipped with several mechanisms with which a large graph is automatically divided into several small sub-graphs. Moreover, SIGPS has several components that are specifically designed for accelerating PageRank processing, such as communication, synchronization and auto GPU execution configuration. Last but not least, the generic API provided by SIGPS make PageRank algorithm composition easier than the others. In Figure 4.18(b), when graph vertex size varies from 100 to 500, both GPU-accelerated methods have an increasing speedup curve. SIGPS has a lower speedup while the graph is not too large. The curve of pure CUDA routine stops at around 500 vertices point while the curve of SIGPS can continue to grow.

4.8.7 Computing Capability Study

PageRank computation is a non-trivial task. The biggest challenge a system encounters is that the input data is extremely huge. To study the extent of the computing capability of SIGPS, we try to generate synthetic graphs that can exhaust the system resources such as main memory, GPU memory or thread pools. Since main memory is much larger than GPU memory and the thread pools are pre-organized as fixed-size blocks and grids, we found that GPU memory is the primary bottleneck that constraints the scalability of the system. Figure 4.19 displays the results of our computing capability study of SIGPS. More specifically, Figure 4.19(a) shows the GPU global memory usage of SIGPS while the graph size (vertex size) increases from 1000 to 10^7. We can see the GPU global memory taken by the graph rises exponentially. When the vertex size arrives 10^7, the graph size cost nearly 4 Gigabytes, which exhausts the GPU global memory. Meanwhile, the global memory available decreases rapidly, the linespoints of which are marked by the inverted triangles. Since SIGPS divides a large graph into subgraphs automatically, we can see in Figure 4.19(b) that the number of subgraphs that SIGPS creates grow exponentially as well. The more partitions a graph is divided, the more replication of the edges are generated. Then the total memory space the graph needs increases.



(a) Memory Usage



(b) Number of Subgraphs

Figure 4.19: Computing Capability Study

4.9 Summary

In this chapter, we present SIGPS, an iterative graph processing model on GPU-accelerated personal computing system. We propose a generic vertex API for users to implement their graph algorithms. By automating GPU execution configuration and parallelizing GPU device functions, we simplified GPU programming for users. Furthermore, emulating shared memory model is designed for vertex communication and data movement within GPU memory. We have conducted extensive experiments to show the effectiveness and efficiency of our system.

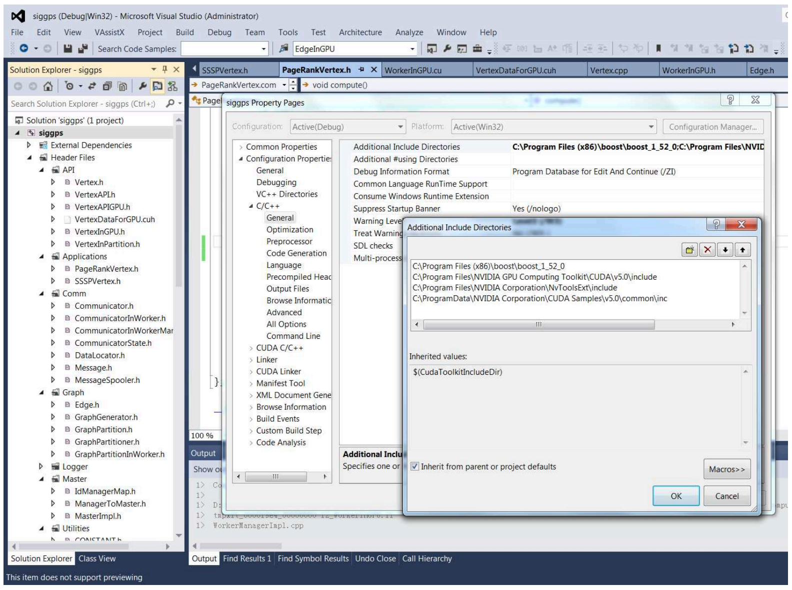
4.10 Appendix

4.10.1 System Installation

To install the SIGPS system, we need a computer system equipped with a CUDA-enabled NVIDIA GPU. Besides, an official CUDA toolkit and the boost library are also required to be installed in the system.

Windows Operating System

In windows, we recommend users to use Microsoft Visual Studio as the programming and compiling platform. In this document, we will use MSVS as an example to introduce how to set up the system.


siggps (Debug|Win32) - Microsoft Visual Studio (Administrator)

siggps Property Pages — Configuration: Active(Debug) — Platform: Active(Win32)

Additional Include Directories	C:\Program Files (x86)\boost\boost_1_52_0;C:\Program Files\NVID
Additional #using Directories	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)

Additional Include Directories

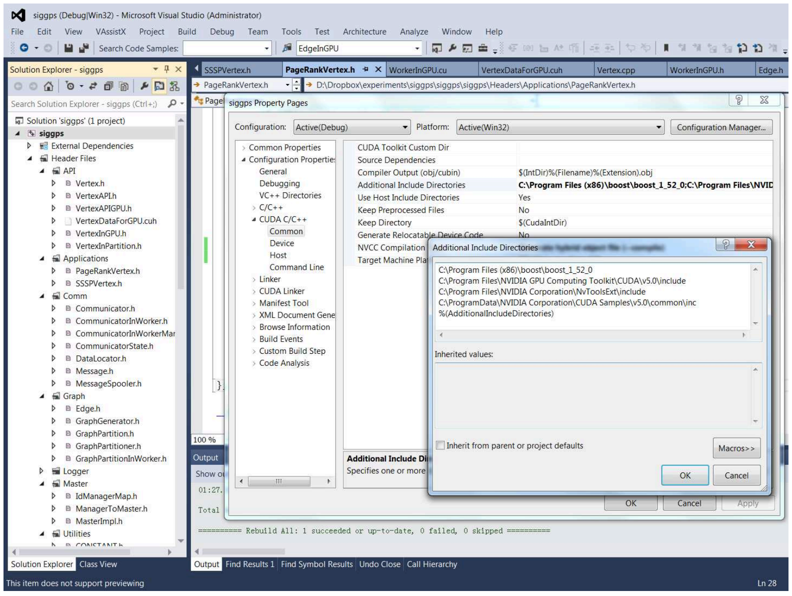
```
C:\Program Files (x86)\boost\boost_1_52_0
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include
C:\Program Files\NVIDIA Corporation\NvToolsExt\include
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\common\inc
```

Inherited values:

```
$(CudaToolkitIncludeDir)
```

☑ Inherit from parent or project defaults

Figure 4.20: Additional Include Directories


siggps (Debug|Win32) - Microsoft Visual Studio (Administrator)

D:\Dropbox\experiments\siggps\siggps\siggps\Headers\Applications\PageRankVertex.h

siggps Property Pages — Configuration: Active(Debug) — Platform: Active(Win32)

CUDA Toolkit Custom Dir	
Source Dependencies	
Compiler Output (obj/cubin)	$(IntDir)%(Filename)%(Extension).obj
Additional Include Directories	C:\Program Files (x86)\boost\boost_1_52_0;C:\Program Files\NVID
Use Host Include Directories	Yes
Keep Preprocessed Files	No
Keep Directory	$(CudaIntDir)
Generate Relocatable Device Code	No

Additional Include Directories

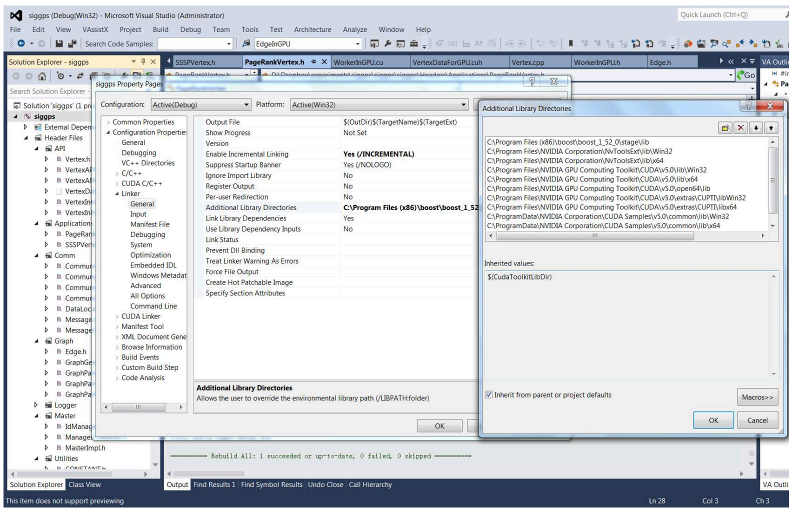
```
C:\Program Files (x86)\boost\boost_1_52_0
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include
C:\Program Files\NVIDIA Corporation\NvToolsExt\include
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\common\inc
%(AdditionalIncludeDirectories)
```

Inherited values:

☐ Inherit from parent or project defaults

```
========== Rebuild All: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Figure 4.21: CUDA Additional Include Directories



siggps Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

```
Output File                        $(OutDir)$(TargetName)$(TargetExt)
Show Progress                      Not Set
Version
Enable Incremental Linking         Yes (/INCREMENTAL)
Suppress Startup Banner            Yes (/NOLOGO)
Ignore Import Library              No
Register Output                    No
Per-user Redirection               No
Additional Library Directories     C:\Program Files (x86)\boost\boost_1_52
Link Library Dependencies          Yes
Use Library Dependency Inputs      No
Link Status
Prevent Dll Binding
Treat Linker Warning As Errors
Force File Output
Create Hot Patchable Image
Specify Section Attributes
```

Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)

Additional Library Directories

```
C:\Program Files (x86)\boost\boost_1_52_0\stage\lib
C:\Program Files\NVIDIA Corporation\NvToolsExt\lib\Win32
C:\Program Files\NVIDIA Corporation\NvToolsExt\lib\x64
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\Win32
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\lib\x64
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\open64\lib
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\extras\CUPTI\libWin32
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\extras\CUPTI\libx64
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\common\lib\Win32
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\common\lib\x64
```

Inherited values:

```
$(CudaToolkitLibDir)
```

[✓] Inherit from parent or project defaults

```
========== Rebuild All: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

siggps (Debug|Win32) - Microsoft Visual Studio (Administrator)
File Edit View VAssistX Project Build Debug Team Tools Test Architecture Analyze Window Help
SSSPVertex.h
PageRankVertex.h
WorkerInGPU.cu
VertexDataForGPU.cuh
Vertex.cpp
WorkerInGPU.h
Edge.h
Macros>>
OK
Cancel
Output Find Results 1 Find Symbol Results Undo Close Call Hierarchy
Ln 28 Col 3 Ch 3

Figure 4.22: Additional Library Directories

After installing the CUDA toolkit and boost library, users can create a new CUDA project (by setting up the CUDA compiling rules) and add the source code of SIGPS into the project. Then users need to open the project property pages under the solution explorer and add the boost library and the CUDA toolkit installation path under the path: “Configuration Properties -> C/C++ -> General -> Additional Include Directories” and the path: “Configuration Properties -> CUDA C/C++ -> Common -> Additional Include Directories”. Moreover, the paths to the libraries of boost library and the CUDA tookit are also needed in the path: “Configuration Properties -> C/C++ -> General -> Additional Libraries Directories”. Figure 4.20, 4.21 and 4.22 are instances of how to setup project environments.

After setting up the project environment, users can add a new class file under “Header Files/Applications” foler. For example, in order to implement the PageRank algorithm in SIGPS, users can add a new vertex class file named “PageRankVertex.h” and implement their PageRank algorithm into the extended Vertex class and the derived virtual class member “Compute()”.

Chapter 5

Asynchronous Iterative Graph Processing System on GPU

In this chapter, we are going to design an asynchronous model of computation on GPU. An update function is able to use the most recent values of the edges and the vertices. The scheduling of all updates is dynamic. The parallel sliding windows method designed in this work implements the asynchronous model on GPU and exposes updated values immediately to subsequent computation. Furthermore, a generic GPU-accelerated graph processing system, ASIGPS, is implemented to support asynchronous concurrency on GPU hardware. Optimized C++ execution engine leverages extensive multi-threading and asynchronous IO. A new software hierarchy is designed to provide better encapsulation and higher modularity. Update/Sync operations are designed to have higher flexibility in synchronous and asynchronous GPU programming.

5.1 Problem Statement

Current graph systems are able to scale to graphs of billions of edges by distributing the computation. To use existing graph frameworks, one is faced with the challenge of partitioning the graph across cluster nodes. Finding efficient graph cuts that minimize communication between nodes, and are also balanced, is a hard problem. More generally, distributed systems and their users must deal with managing a cluster, fault tolerance, and often unpredictable performance. From the perspective of programmers, debugging and optimizing distributed algorithms is hard.

Since SIGPS is based on **Bulk Synchronous Parallel** (BSP) Model, it is also constrained by the limitation of BSP model. SIGPS execute all vertex **compute** functions in lock step, which can only observe values from its previous iteration. BSP is often preferred in distributed systems as it is simple to implement, and allows maximum level of parallelism during the computation. However, after each iteration, a costly synchronization step is required and system needs to store all values of both iterations.

There are some obvious drawbacks for synchronous computational model. On one hand, the synchronous abstraction forces some cohesively connected algorithms to be torn down into parts and transformed into an embarrassingly parallel computations. Additionally, synchronization also forces most of the fast threads to wait for individual slowest one, which greatly harness the processing speed. On the other hand, though some parallel graph algorithms are inherently designed to be synchronous, most iterative graph processing algorithms are difficult to be executed synchronously after parallelizing. In some especial cases, BSP fails to converge at all, which will block all the threads from executing. In addition, most of the current graph processing systems base on the BSP model and do not support any asynchronous processing, no mention asynchronous computation on GPUs. On GPUs, synchronous processing executes all device functions concurrently and parameters for each of them need to be prepared before the computation.

Conversely, asynchronous processing do not need to tear apart or transform cohesively connected algorithms. No thread need to wait for each other at the end of every single iteration. They can run as fast as they can. Moreover, GPU hardware may execute device functions in any style. The functions can use the most-recent-generated parameters. Asynchronous computation accelerates convergence of many numerical algorithms.

Another problem that needs to be mentioned is that it is challenge to store a large graph in main memory or GPU memory during processing. Furthermore, efficient modification of the edge values demand the capability of random access, which is also problematic in most of the current systems. Advanced storage media such as SSD can only support tens of thousands of random reads/writes per second, while millions of access may be required for a huge graph. Exploiting graph locality is also limited due to the skewed vertex degree distribution. System performance is unpredictable as it depends on the graph structure. In ASIGPS, we solved the random access problem by writing updated edges into a scratch file, which is then disk-sorted, and used to generate input graph for next iteration. However, this method cannot be efficiently used to perform asynchronous computation. Therefore, an efficient and compressed data format and a new access scheme are in need for our asynchronous graph processing system.

5.2 Graph Formats for Asynchronous Computing on GPU

5.2.1 Compressed Row/Column Storage on GPU

The GPU-accelerated system stores the graph on disk and transfers it to be processed in GPU memory. An efficient data format is required to reduce data transfer time and space cost. It should be equivalent to adjacency formats and edges need to be stored consecutively in the memory address or file. In addition, edges or out/in edges of one vertex is required to be loaded fast. In order to transfer graphs across disk, main memory and GPU memory, we employed an compressed graph format in this work.

The Compressed Row Storage on GPU (CRSG) format puts the subsequent non-zeros of the matrix rows in contiguous GPU global memory locations. Assuming we have a non-symmetric sparse matrix A, we create three vectors: one for floating point numbers (val) and the other two for integers (col_ind, row_ptr). The val vector stores the values of the non-zero elements of the matrix A as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. That is, if $\texttt{val(k)} = a_{i,j}$, then $\texttt{col\_ind(k)} = j$. The row_ptr vector stores the locations in the val vector that start a row; that is, if $\texttt{val(k)} = a_{i,j}$, then $\texttt{row\_ptr(i)} \le k < \texttt{row\_ptr(i + 1)}$. By convention, we define $\texttt{row\_ptr(n + 1)} = nnz + 1$, where nnz is the number of non-zeros in the matrix A. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2nnz + n + 1$ storage locations.

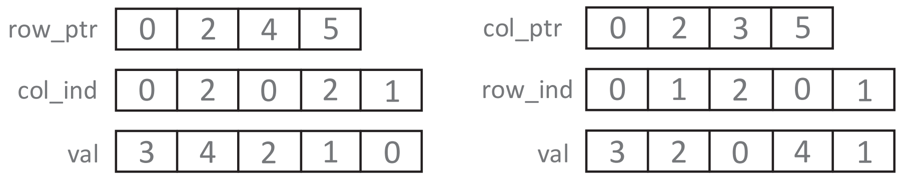
Example 6. *As an example, consider the non-symmetric matrix A defined by:*

$$\begin{pmatrix} 3 & & 4 \\ 2 & & 1 \\ & 0 & \end{pmatrix}$$

The CRSG format for this matrix is then specified by the arrays $\{val, col\_ind, row\_ptr\}$ given below in Figure 5.1(a).

Analogous to CRSG, there is Compressed Column Storage on GPU (CCSG), which is also called the Harwell-Boeing sparse matrix format on GPU. The CCSG format is identical to the CRS format except that the columns of A are stored (traversed) instead of the rows. In other words, the CCSG format is the CRS format for A^T.

The CCSG format is specified by the 3 arrays $\{val, row\_ind, col\_ptr\}$, where row_ind stores the row indices of each non-zero, and col_ptr stores the index of the elements in val *which start a column of A. The CCSG format for the matrix A is given by Figure 5.1(b).*



(a) CRSG (b) CCSG

Figure 5.1: Compressed Graph Storage on GPU

5.3 Asynchronous Computational Model

A directed graph $G = (V, E)$ is named as a directed graph. Each vertex $v \in V$ and each directed edge $e = (source, destination) = (u, v) \in E, 1 \leqslant u, v \leqslant |V|$ are associated with a value respectively. An update function is designed to access and modify the value of a vertex and its incident edges. The update function is executed for each vertex iteratively before a termination condition is reached.

Algorithm 22: Generic API For User Derived Vertex Update Function

```
1 __device__ void update(v) {
2         Read values of all edges of vertex v ;
3         Compute the new value for vertex v ;
4         Compute and assign the new values for all edges of vertex v ;
5 }
```

The asynchronous computational model on GPU is implemented by way of Parallel Sliding Windows on GPU. We will introduce Parallel Sliding Windows on GPU in the following section. Algorithm 22 is the generic API for user derived vertex update function. The underlying three lines in this function describe the main tasks executed by this GPU device function. Algorithm 23 is an example of the aforementioned “update” function, which is described in pseudo-code. All edge values are read into some array in

Algorithm 23: Pseudo-code for Vertex Update Function

```
1 __device__ void update(v) {
2     for ( all edges e of vertex v ) {
3         Array[e.index] = e.value ;
4     }
5     v.value = f( Array[] ) ;
6     for ( all edges e of vertex v ) {
7         e.value = g( v.value(), e.value() ) ;
8     }
9 }
```

line 3. Function f executes computation over the array and stores it as the vertex value in line 5. From line 6 to 8, the vertex value is spread to all its neighboring edges, if some condition coded within function g is satisfied.

5.4 Parallel Sliding Windows on GPU

Parallel Sliding Windows on GPU (PSWG) can process a graph with mutable edge values efficiently from disk through main memory, with only a small number of non-sequential disk accesses and memory transactions. PSWG supports the asynchronous model of computation. There are five steps for PSWG to process a graph:

1. a subgraph is loaded from disk into main memory;
2. the subgraph is transferred to GPU global memory;
3. the vertices and edges are updated;
4. the subgraph is transferred back to main memory;
5. the updated values are written to disk.

We then explain the aforementioned steps in the following sections.

5.4.1 Loading the Graph From Disk to GPU global memory

Using the PSWG method, the vertices V of graph $G = (V, E)$ are split into N disjoint segment. For each segment, we associate a block, which stores all the edges that have destination in the segment. Edges are stored in the order of their source. Segments are chosen to balance the number of edges in each block; the number of segments, N, is chosen so that any one block can be loaded completely into GPU global memory. PSWG does graph computation by processing vertices one segment at a time. To create the subgraph for the vertices in segment i, their edges must be loaded from disk.

As is shown in Figure 5.2, block(i) contains the in-edges for the vertices in segment(i) and is loaded fully into GPU global memory. We call such block(i) the memory-block. Because the edges are ordered by their source, the out-edges for the vertices are stored in consecutive chunks in the other blocks, requiring additional $N - 1$ data reads. Importantly, edges for segment(i+1) are stored immediately after the edges for segment(i). Intuitively, when PSWG moves from a fragment to the next, it slides a window over each of the blocks. We call the other blocks the sliding window blocks. Figure 5.3 illustrates the process of loading the graph. Vertices of the graph are divided into four fragments. there are one block linked to each fragment. PSWG constructs one subgraph for each fragment. One fragment in dark is first stored in GPU global memory and all in-edges for the vertices in this fragment can be read directly. All out-edges are read in the sliding window blocks from the disk.

5.4.2 Parallel Updates

Parallel Sliding Window on GPU (PSWG) executes the user-defined update-functions for each vertex in parallel when the subgraph for segment i has been fully loaded from disk. To prevent race conditions (one edge value was read by two end vertices at the same time), we enforce that each execution of PSWG would generate the same value.

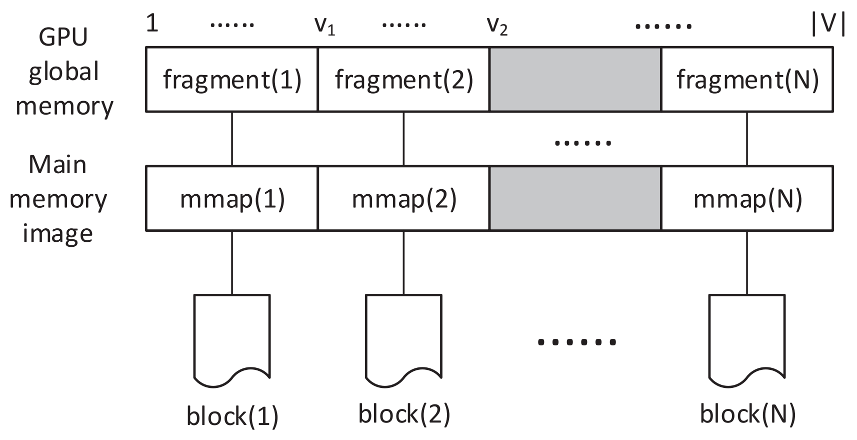


Figure 5.2: PSWG Block Mapping

As is noted in Definition 3.5.1, only cut-edges can be updated in parallel. According to the asynchronous model of computation, all inner vertices and peripheral vertices are considered as critical vertices. Each critical vertex will notice the preceding updates of other critical vertices that are connected to it. Therefore, for algorithms that demand consistency, critical vertices are executed in sequential order.

5.4.3 Updating Graph to Disk

After all vertices and edges are updated, PSWG writes them back to disk, replacing the old data. Actually, PSWG loads the edges from disk in large blocks. The edge blocks are cached in main memory before being sent to GPU global memory. When the fragment subgraph is created, the edges are referenced as pointers to the cached blocks; modifications to the edge values directly modify the data blocks themselves. The active block and the active sliding window of other blocks are written to disk. Then PSWG moves to the next fragment and reads the new blocks from disk. As is mentioned above, the number of parallel disk writes for a fragment is N.

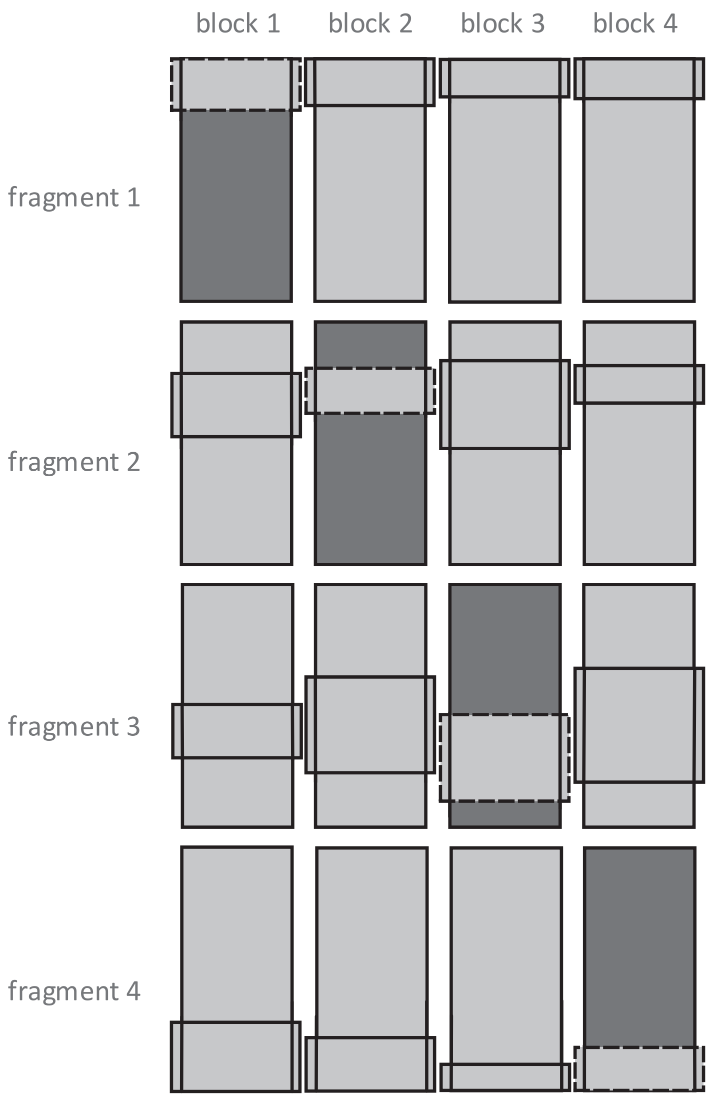


Figure 5.3: PSWG Sketch

5.5 System Design and Implementation

5.5.1 Block Graph Data Format on GPU

The block graph data format on GPU is designed as an efficient and compressed format for storing the blocks on GPU. Since most of the graph mining algorithms work on graphs with a fixed structure and update the edge data only. We may separate the graph structure from its associated edge values. Besides exploiting the sparsity of the graph, we

Algorithm 24: Parallel Sliding Windows on GPU (PSWG)

```
for( it = iteration.begin(); it != iteration.end(); ++it ) {
    Initialize(fragments);
    for( frag = fragments.begin(); frag != fragments.end(); ++frag ){
        loadSubgraphToGPU(frag, subgraph) ;
        updateVertexOnGPU() ;
        loadSubgraphFromGPU(frag, subgraph) ;
        subgraph.blocks[frag].UpdateFully() ;
        for ( ix = fragments.begin(); ix != fragments.end(); ++ix ) {
            if(ix != frag)
                subgraph.blocks[frag].UpdateLastWindow() ;
        }
    }
}
```

can generate and access a graph both on GPU and disk. There are two block components currently, namely adjacency block and edge data block. The adjacency block records the neighboring information in an order of a concatenate edge array and an index array. The edge data block is an array of edge values.

5.5.2 Preprocessing

After reading it from the disk, the system preprocess the graph to generate graph fragments before any further computation. The preprocessor first scans the graph and computes the degree of the vertices by the prefix sum method. Then the preprocessor re-order the vertices and divide vertices into N segments. These segments are constrained with nearly the same degree sum. After that, the preprocessor runs over the graph file for the second pass and writes each edge to a scratch file for each fragment. The processor sorts the edges in each scratch file and writes them to the compact block files. In order to assist efficient computation on GPU, the preprocessor generate a temporary binary degree file that stores the degrees for each vertex.

5.5.3 Execution

After preprocessing the raw graph data, the system iteratively processes the fragment subgraphs. Figure 5.4 illustrates the processing iteration for one execution fragment. Loading degree data for the fragment, the system also preallocates edge arrays and vertex objects in main memory. Then the system executes PSWG algorithm to load fragment data from the disk to GPU global memory. After launching a batch of GPU threads to execute vertex compute functions in parallel. Symmetrically, the system transfers the updated blocks back to the disk before going to the next fragment.

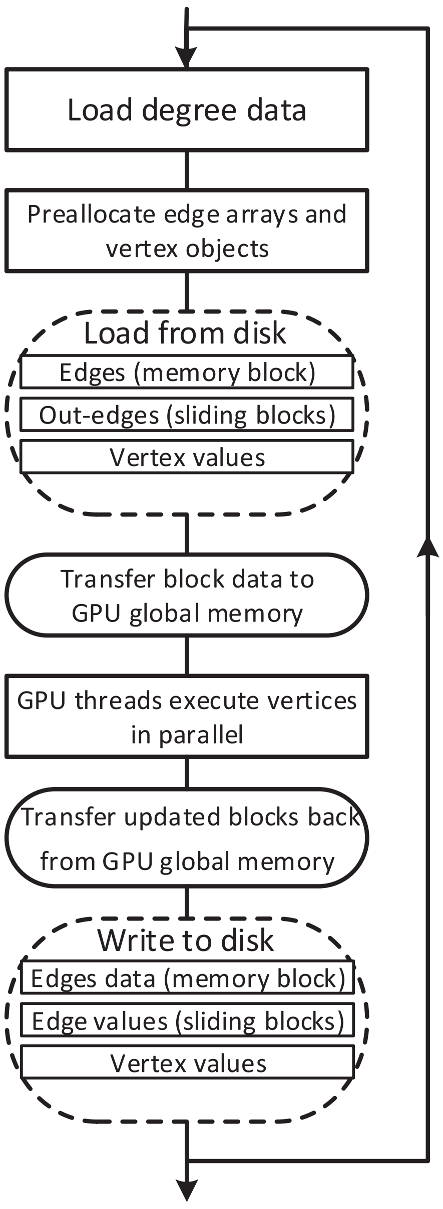

Load degree data
Preallocate edge arrays and vertex objects
Load from disk
Edges (memory block)
Out-edges (sliding blocks)
Vertex values
Transfer block data to GPU global memory
GPU threads execute vertices in parallel
Transfer updated blocks back from GPU global memory
Write to disk
Edges data (memory block)
Edge values (sliding blocks)
Vertex values

Figure 5.4: Execution Flow

5.5.4 Software Hierarchy Overview

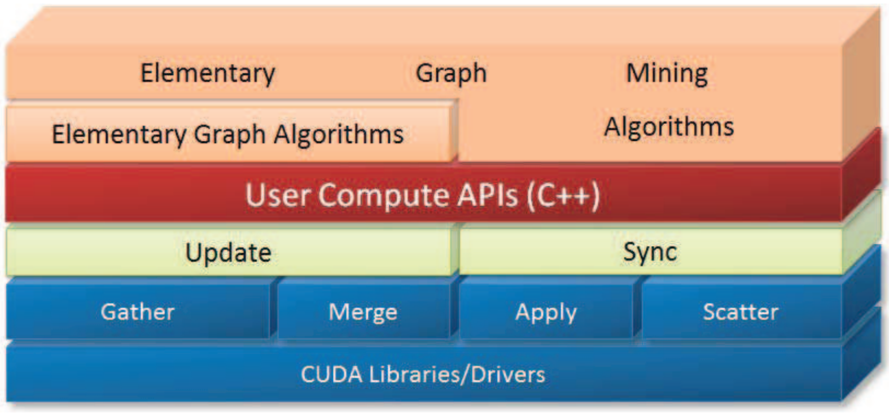


Figure 5.5: Software Hierarchy

ASIGPS is a generic graph processing system that supports asynchronous computation on GPU. Figure 5.5 shows the ASIGPS software hierarchy that consists of five main layers. The top tier contains elementary graph algorithms and elementary graph mining algorithms that are pre-implemented as a library. The second layer is the user compute APIs within which users can implement their own algorithms. The layer below are the two lower-APIs, “Update” and “Sync”, which can be utilized to implement programmable behaviors for vertices. Below “Update” and “Sync”, there are four collective operations implemented for GPU computations. They are “Gather”, “Merge”, “Apply” and “Scatter”. The substratum is the CUDA libraries and drivers that are the basic functional component of a GPU-accelerated system.

5.6 Programming Model and Application Programming Interfaces

Programs written for ASIGPS are similar to those written for SIGPS. However, the differences are as follows. First, SIGPS is based on Bulk Synchronous Parallel (BSP model, while ASIGPS do not need to synchronize at the end of each iteration. Second, SIGPS emulate message passing for communications among vertices while ASIGPS removes this model and updates the edge values directly.

Algorithm 25 is the generic API for users to program their own algorithms. User would implement a class that derives their own methods that inherited from ASIGPS vertex template. During execution, ASIGPS would automatically launch a batch of GPU threads to execute vertex computes. Within the vertex “compute()” function, users can implement an asynchronous computation via “update()” function and apply necessary synchronization via “sync()” function.

Algorithm 25: Generic API For User Derived Program Class

```
class DerivedVertex : AsigpsVertexInGPU {
public:
    __device__ void compute() {
        ...  // Algorithm implemented by users.
    }
    ...  // Other public members can be added by users.
private:
    ...  // Private members can be added by users.
} ;
__device__ VertexRegisterInGPU<DerivedVertex> reg ;
```

Moreover, there are four collective operations are provided by ASIGPS that can be invoked by users within “update()” and “sync()” functions. Algorithm 26 shows the device function “gather” for the current vertex thread to gather the related values from its neighbors. “gather” function stores the corresponding values in an array for further processing.

Algorithm 26: Generic API For Function Gather

```
__device__ void gather(vertex) {
        for ( e in vertex.inEdges() ) {
                neighborValues[“e”] = e.getValue() ;
        }
        vertex.setNeighborValues(neighborValues) ;
}
```

Algorithm 27 displays the device function “merge”, which reduces all values gathered from the vertex’s neighboring edges.

Algorithm 27: Generic API For Function Merge

```
__device__ void merge(vertex) {
        for ( e in vertex.inEdges() ) {
                sum += e.getValue() ;
        }
        vertex.setValue(sum) ;
}
```

Algorithm 28 is the device function “apply”, which updates the vertex value using “function” and scatters this updated value to all out-edges.

Algorithm 28: Generic API For Function Apply

```
__device__ void apply(vertex, (*f)(double)) {
        for ( e in vertex.outEdges() ) {
                e.setValue((*f)(vertex.getValue)) ;
        }
}
```

Algorithm 29 shows the device function “scatter”, which collectively sends the corresponding values to all out-edges from the vertex neighboring-value-array.

Finally, the vertex update function can be written as a composition of “gather”, “apply” and “scatter” functions. This is displayed in Algorithm 30.

Algorithm 29: Generic API For Function Scatter

```
__device__ void scatter(vertex) {
    for ( e in vertex.outEdges() ) {
        e.setValue(vertex.getNeighborValues[“e”]) ;
    }
}
```

Algorithm 30: Vertex Update Function

```
__device__ void update(v) {
    gather(v) ;
    apply(v,(*f)) ;
    scatter(v) ;
}
```

5.7 Case Study and Applications

5.7.1 Case one: PageRank

PageRank is an algorithm used by Google Search to rank websites in their search engine results. It is a way of measuring the importance of website pages and one of many factors used to determine which pages appear in search results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

PageRank is a link analysis algorithm and it assigns a numerical weighting to each element of a hyper-linked set of documents, such as the World Wide Web, with the purpose of “measuring” its relative importance within the set. The computations of the algorithm have several passes, called “iterations” through the collection to adjust approximate PageRank values to more closely reflect the theoretically true value. Generally in mathematics, the PageRank value for any page **u** can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)} \approx 0.15 \times \frac{1}{|V|} + 0.85 \times \sum \frac{PR(v)}{L(v)}$$

To implement the PageRank algorithm using ASIGPS, we extend the Vertex class on the GPU to create a **PageRankVertex** user-defined GPU class. The PageRankVertex class derives the **compute()** virtual method, executed by one GPU thread, to calculate the PageRank value of the vertex.

Algorithm 31: PageRankVertex

```
1  class PageRankVertex : VertexInGPU {
2  public:
3      __device__ void compute() {
4              double sum = 0.0 ;
5              for ( e in vertex.inEdges() ) {
6                  sum += e.getValue() ;
7              }
8              int vnum = this->getTotalNumVertices() ;
9              double newPageRank = 0.15 / vnum + 0.85 * sum ;
10             this->setValue(newPageRank) ;
11
12             for ( e in vertex.outEdges() ) {
13                 e.setValue(vertex.getValue) ;
14             }
15     }
16 };
17
18 __device__ VertexRegisterInGPU<PageRankVertex> reg ;
```

Algorithm 31 is an example of user-defined PageRankVertex class implemented for asynchronous GPU execution. We take advantage of the generic Vertex APIs to derive the function “compute()”. In line 5 to 7, the GPU thread reads its neighboring edge values directly and sums them up. A new PageRank is calculated in line 9 and updated for the vertex value in line 10. In line 12 to 14, ASIGPS spreads this updated value to all its neighbors. Similar to the situation in SIGPS, users need to declare a global variable, instantiated by the class name **"PageRankVertex"** in the last line, to register this customized vertex class in ASIGPS.

Algorithm 32 is another version of user-defined PageRankVertex class. In this algorithm, we take advantage of the collective operations provided by ASIGPS. It is noticeable that this algorithm is so simplified that we can write it in only 4 lines.

Algorithm 32: PageRankVertex

```
class PageRankVertex : VertexInGPU {
public:
    __device__ void compute() {
            int vnum = this->getTotalNumVertices() ;
            auto calPageRank = [&](x){ return 0.15 / vnum + 0.85 * x ; }
            merge((*this)) ;
            apply((*this), calPageRank) ;
    }
};

__device__ VertexRegisterInGPU<PageRankVertex> reg ;
```

5.7.2 Application

We also implemented algorithms for several other applications, such as SSSP, Dense Graph Mining and triangle counting. The first algorithm is the problem of finding a path between vertices in a graph so that the sum of the weights of the edges in the path is minimized. The algorithm converges when all the vertex values are set to be the shortest distances to the source vertex. There are several iterations before the algorithm converges. A temporary shortest distance and a variable pre-node are utilized to record the tentative information. Each vertex starts with an initial value. The vertex threads read their neighboring edge values directly. If the tentative shortest distance is smaller than the edge value fetched, the tentative distance will be replaced by this smaller one and its pre-node information will be updated as well. After accessing all neighboring edge values, the shortest tentative distance will be compared with the vertex value and the smaller one will be retained as the updated vertex value. ASIGPS will compare all vertex values and their connected edge values in parallel asynchronously on GPU. Once a vertex updates its value, it would spread to all its neighboring edges and trigger others to update theirs. The second algorithm is based on label propagation. At the beginning, each vertex writes its id (“label”) to its edges. Then vertex chooses the most frequent value (“label”) according to its neighboring edge values (“labels”). ASIGPS schedules a vertex only if the value (“label”) in a connecting edge is updated. Vertices with the same value (“label”) are regarded as the the connected dense subgraph. The third algorithm is to count the number of edge triangles incident to every vertex. To efficiently join the neighbors of two vertices, the graph is re-ordered according to vertex degree. A subgraph fragment with higher degrees are stored in GPU global memory and other fragments are then read from the disk for comparison.

5.8 Performance Comparison with SIGPS

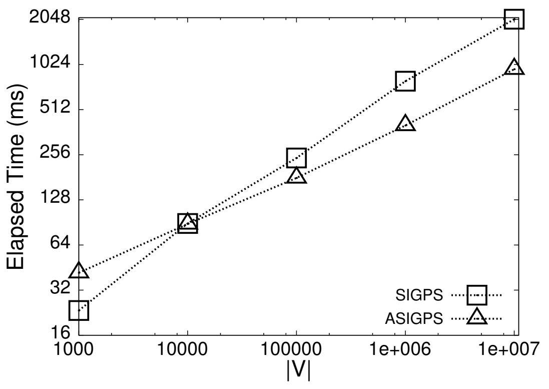
We have conducted our main performance comparison experiments on the desktop equipped with an NVIDIA GeForce GTX 760 graphics card. The desktop is driven by a 4-core i7-4770 x64-based central processor (8M Cache, 3.40 GHz). The graphics processing unit has 6 multi-processors, each of which has as many as 192 processing cores (8M Cache, 1.15 GHz). The main memory is 16GB while the GPU global memory is 4GB. The PageRank algorithms implemented for SIGPS and ASIGPS are executed on the massive graphs to compare the performance and scalability.

We employ synthetic and real datasets in this study. Experimental synthetic graphs are generated by the system graph generator component. A series of graphs with varying vertex sizes from 10^3 to 10^7 are created. Real graphs include flickr, DBLP, PPI, and

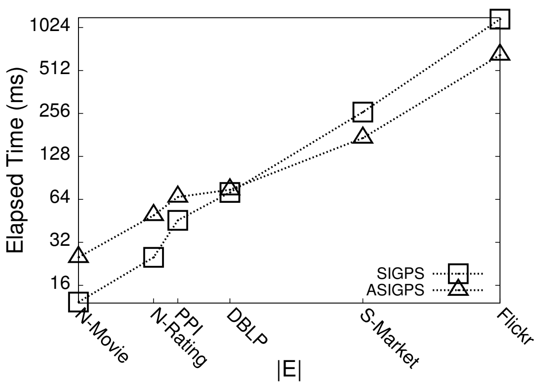
Netflix datasets. Flickr graph is derived from a well known photo sharing social network. There are 1,715,255 people and 22,613,982 sharing relationship recorded in the Flickr graph. The DBLP dataset records 23136 authors and their 54989 co-authorship. The Protein Protein Interaction (PPI) graph contains 17203 interactions among 4930 proteins, which records the behavior of the entire interactomics system of a living cell. There are 480,000 customers and 17,000 movies in the Netflix datasets, which are generated from an American on-demand internet streaming media.

5.8.1 Scalability

So as to compare the scalability of SIGPS and ASIGPS, we run the PageRank algorithms on synthetic and real graphs with increasing sizes. Figure 5.6 illustrates the growing tendency of the running time of the corresponding algorithms. When graph size increases exponentially, the elapsed time of processing a graph rises up with an accelerated speed. More specifically, in Figure 5.6(a) PageRank algorithms are executed on synthetic graphs with vertex size ranging from 10^3 to 10^7. We can observe that when a graph is small, system SIGPS runs faster than ASIGPS. This is because ASIGPS takes longer time to prepare data before the algorithm is executed. While the graph size increases, more GPU threads are employed to operate concurrently. There is no need for threads of ASIGPS to wait for each other between consecutive iterations, while threads of SIGPS are forced to wait for each other by obvious barriers. Therefore, as graph size increases, PageRank for ASIGPS runs faster and faster than that for SIGPS. Similarly, Figure 5.6(b) displays the increasing tendency of the elapsed time of the PageRank algorithm executed on several real graphs. ASIGPS runs faster than SIGPS on real graphs when graphs are large enough. We notice that SIGPS runs faster on the Netflix N-Movie and N-Rating graphs, PPI graph, and DBLP graph, while ASIGPS has better performance on the larger graphs, S-Market and Flickr.



(a) Synthetic Graphs



(b) Real Graphs

Figure 5.6: Execution Time

5.8.2 Data Communication

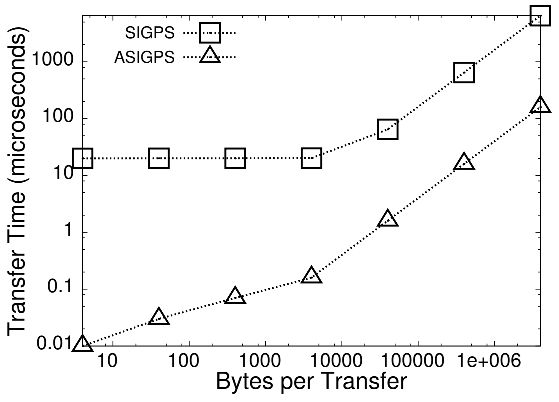


Figure 5.7: Communication Cost

To compare the communication cost of the two systems, we study the data movement, calculate the communication throughput and plot them in Figure 5.7. We increase message size and compare data moving in the two systems. Figure 5.7 shows the data transfer time when the updated edges/messages size is increased. SIGPS uses message passing mechanism that transfers updated “messages” to main memory while ASIGPS directly writes edges to GPU global memory. In the figure, SIGPS takes more communication cost than ASIGPS. We can observe that a message transfer to main memory uses around 20 microseconds. When the data transferred is too large, it is packed in several messages and are sent in queue, which increases the total cost. The turning point of ASIGPS curve means the system start to transfer edges to main memory from the GPU global memory.

5.8.3 Speedup

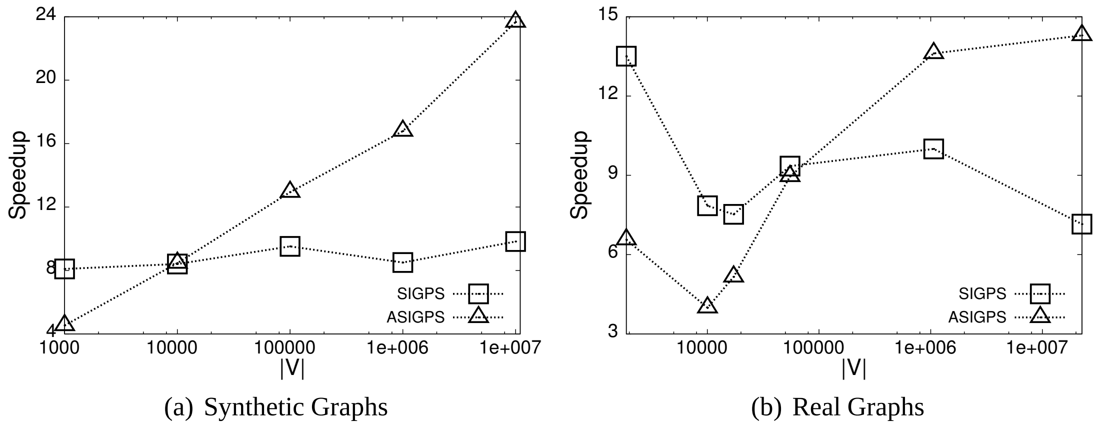


Figure 5.8: Speedup

To compare the speedup of SIGPS and ASIGPS, we apply the PageRank algorithms on the two systems respectively. The PageRank algorithms are executed concurrently by thousands of GPU threads and the sequential mode of SIGPS is set to be the baseline. Figure 5.8 displays the comparison of the speedups when the algorithms are run on synthetic and real graphs. More specifically, Figure 5.8(a) illustrates the speedups when the algorithms run on synthetic graphs. From the plot, the speedup curve for SIGPS are steady and that for ASIGPS has an increasing tendency. When the graph size (vertex size) is smaller than 10000, SIGPS has a higher speedup than ASIGPS. The ASIGPS has the burden of preparing data and low parallelism makes the benefit of the asynchronization ineffective. While the graph size increases, the speedup of ASIGPS goes up accordingly. Figure 5.8(b) displays the speedups when the algorithms are executed on real graphs. Similarly, when the graph size (edge size) is smaller than around 60000, SIGPS has a higher speedup. And while the graph grows larger, ASIGPS takes the lead in speedup and performance.

5.9 Summary

In this chapter, we proposed ASIGPS, an asynchronous iterative graph processing model on GPU-accelerated personal computer system. ASIGPS was designed as an alternative to SIGPS. In this chapter, we proposed an asynchronous computation model, PSWG, on GPU. We designed new graph formats for asynchronous computing on GPU. A set of generic APIs are also provided for users to implement their own algorithms. Collective GPU operations are also provided for efficient GPU programming. As a generic graph processing model on GPU, ASIGPS is both sufficiently expressive to implement a wide range of graph processing algorithms, and formidably powerful to drive efficient large graph processing.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude this thesis and address some future work on the basis of the proposed graph processing model/system and methods in this thesis. Specifically, Section 6.1 provides a brief summary on the contributions of the thesis. Section 6.2 formalizes a few promising research directions and applications to extend our current studies.

6.1 Summarization

This thesis focuses on utilizing GPGPU techniques over large graph mining problems. While traditional processing techniques are only applicable to the graphs of limited sizes on general computer systems, all of these techniques processing graphs exceeding specific sizes encounter bottlenecks in the system, when computing power is no more enough and graphs are too big to be stored in the memory. These problems prohibit the use of efficient graph processing algorithms on the general computer systems with quickly evolving large graphs.

The state-of-the-art GPGPU techniques are utilizing many-core graphics processors to perform general purpose computation. It was found that GPGPU techniques greatly accelerate graph triangulation algorithm. Comparing with the methods provided by Wang [51], the speedup gained by GPU-accelerated triangulation is around 5 to 20, which is quite remarkable (Chapter 3). Triangulation normally functions as a basic approximative module for dense graph mining. A possible explanation is that the application of SIMD multi-threading model on many-core GPUs extremely extends the inherent parallelism of the graph and algorithm. This result suggests that GPGPU techniques can be employed to accelerate graph mining algorithms. The work in this thesis is the first attempt to accelerate graph triangulation using GPGPU techniques. The finding is significant for personal computers as it provides a potential solution for large scale domain applications, which previously can only be processed by main-frame/distributed systems.

After finding the methods for breaking the system bottlenecks, we opt for a systematic and generic solution for efficient and economic large graph processing. Therefore, a synchronous graph processing model over GPU-accelerated platform was designed in Chapter 4 and a generic graph processing system was built on this model. The main difference between the model/system here and the existing graph processing library is that a set of generic APIs are provided for assisting users to compose their own algorithms. Using the template of this model/system, existing or user-defined graph mining algorithms, including those of massive domain applications, can be easily implemented on top of general computer systems with limited resources. Moreover, GPU execution configuration/process is automated and transparent to users. Flexible threading mechanism and hierarchical module architecture have given the system high extendibility and scalability. This system can bring an impressive impact over the graph mining community.

However, the synchronization exerted by the model forces all vertices represented by the light-weighted GPU threads to wait for each other. Because the degree distributions of these large scale domain graphs are highly skewed, a majority of the vertices with low degrees have to idle for most of the time. This has greatly affected the performance of the system. Therefore an improved model that provided asynchronous computing was then proposed in Chapter 5. The parallel sliding windows on GPU implemented the model and exposed updated values immediately to subsequent computation. Besides the vertex API “compute”, there were two new operational APIs named ”sync” and ”update”. Moreover, four collective GPU operations were provided to assist efficient programming. A new generic graph processing system that supports the asynchronous processing over GPU-accelerated large graph applications was re-designed and implemented. The improved model has successfully brought in the asynchronous computing to graph mining, which greatly improve the performance of the system. This improvement is a significant step for generic graph mining.

6.2 Possible Research Directions and Applications

ASIGPS was designed for asynchronous iterative graph processing, which can be utilized to implement advanced graph mining algorithms. We consider to extend ASIGPS to support dynamic graph mining, which demands millions of vertex updates at the same time. A continuous graph updates, accompanied with concurrent graph-related computation, incurs great challenge for a single personal computer system. Moreover, it is interesting to deploy SIGPS and ASIGPS over distributed GPU-accelerated system. Suitable adjustments to the computation model should support pipelining, multi-layer many-threading asynchronous graph processing. Efficient communication will be a problem in this situation.

It is noted that there may be a few problematic issues involved in the system since designing an effective and efficient system across heterogeneous platform is complicated. More efforts need to be paid to solve all the problems related to the implementation of the hybrid system. Additionally, system optimization can further improve the performance. It is noticed that we have only provided several demonstrative algorithms using the system. More graph mining algorithms need to be implemented to constitute the library of the system. It is also understood that we only focus on graph processing on top of personal computer systems. More data mining applications and graph processing accelerated by connected distributed GPU nodes are very interesting but beyond the scope of this thesis. Further study/research is needed to extend the model/system to support more general data mining applications. This is much more challenging but will bring greater impact to the whole data mining community.

Bibliography

[1] *NVIDIA CUDA Programming Guide*, 2011.

[2] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. LATIN '02, 2002.

[3] Aggarwal and etc. *Managing and Mining Graph Data*. 2010.

[4] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pages 540–549, Washington, DC, USA, 2004. IEEE Computer Society.

[5] AMD. Opencl demo, amd cpu. SIGGRAPH, 2008.

[6] AMD AMD. Close to metal. *Technology Unleashes the Power of Stream Computing. Webpage*, 2006.

[7] Reid Andersen. A local algorithm for finding dense subgraphs. 2007.

[8] Ziv Bar-yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs.

[9] Becchetti and etc. Efficient semi-streaming algorithms for local triangle counting in massive graphs. 2008.

[10] Adam L. Buchsbaum, Raffaele Giancarlo, and Jeffery R. Westbrook. On finding common neighborhoods in massive graphs. *Theor. Comput. Sci.*, 299(1-3):707–718, April 2003.

[11] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[12] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. The MIT Press, 2008.

[13] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, 2000.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Trans. Algorithms*, 6(1):6:1–6:17, December 2009.

[16] Nicholas Edmonds. The parallel boost graph library spawn(active pebbles). *KDT Mind Meld*, 03/2012 2012.

[17] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[18] Joan Feigenbaum, Sampath Kannan, Andrew Mcgregor, and Jian Zhang. On graph problems in a semi-streaming model. In *In 31st International Colloquium on Automata, Languages and Programming*, pages 531–543, 2004.

[19] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, 2005.

[20] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. Mpi: The complete reference, vol. 2—the mpi-2 extensions. *Published in*, 1998.

[21] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. 2007.

[22] Guoming He, Haijun Feng, Cuiping Li, and Hong Chen. Parallel simrank computation on large graphs with iterative aggregation. KDD, 2010.

[23] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on data streams, 1998.

[24] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly, 2013.

[25] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2010.

[26] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. In *Proceedings of the eighth international conference on World Wide Web*, 1999.

[27] Chuck Lam. *Hadoop in action*. Manning Publications Co., 2010.

[28] Jinyan Li, Kelvin Sim, Guimei Liu, and Limsoon Wong. Maximal quasi-bicliques with balanced noise tolerance: Concepts and co-clustering applications. In *SDM'08*, 2008.

[29] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *ECML PKDD*, 2008.

[30] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[31] K. Madduri, D.A. Bader, J.W. Berry, J.R. Crobak, and B.A. Hendrickson. Multithreaded algorithms for processing massive graphs. 2008.

[32] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. 2004.

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[34] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[35] Mattson and etc. *Patterns for parallel programming*. 2004.

[36] Andrew McGregor. Graph mining on streams. In *Encyclopedia of Database Systems*, pages 1271–1275. 2009.

[37] K. Musgrave and University of Wales Swansea. Dept. of Computer Science. *Generic Programming and the Boost Graph Library*. University of Wales Swansea, 2004.

[38] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[39] NVIDIA. Opencl demo, nvidia gpu. SIGGRAPH, 2008.

[40] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[41] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[42] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, 2005.

[43] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. *Cloud Computing Technology and Science, IEEE International Conference on*, pages 721–726, 2010.

[44] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. CLOUDCOM ’10, 2010.

[45] Arun Suresh. Phoebus: Erlang-based implementation of googles pregel, 2010.

[46] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, pages 103–111, 1990.

[47] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. HPG, 2009.

[48] Vibhav Vineet and P. J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. *Computer Vision and Pattern Recognition Workshop*, 2008.

[49] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 2001.

[50] Nan Wang, Srinivasan Parthasarathy, Kian-Lee Tan, and Anthony K. H. Tung. Csv: visualizing and mining cohesive subgraphs. 2008.

[51] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. On triangulation based dense neighborhood graphs discovery. *VLDB*, 2010.

[52] Jianlong Zhong, Bingsheng He, and Gao Cong. Medusa: A unified framework for graph computation and visualization on graphics processors. Technical report, Nanyang Technology University, 2011.