Novel Techniques for Graph Algorithm Acceleration

by Hang Liu

B. E. in Software Engineering, May 2011, Huazhong University of Science & Technology

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of the George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2017

Dissertation directed by

H. Howie Huang
Associate Professor of School of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Hang Liu has passed the Final Examination for the degree of Doctor of Philosophy as of June 12th, 2017. This is the final and approved form of the dissertation.

Novel Techniques for Graph Algorithm Acceleration

Hang Liu

Dissertation Research Committee:

H. Howie Huang, Associate Professor of Engineering and Applied Science, Dissertation Director

Ahmed Louri, Professor of Engineering and Applied Science, Committee Member

Milos Doroslovacki, Associate Professor of Engineering and Applied Science, Committee Member

Guru Venkataramani, Associate Professor of Engineering and Applied Science, Committee Member

Timothy Wood, Associate Professor of Engineering and Applied Science, Committee Member

Dedication

To my beloved wife Ni Yang, Parents Qiming Liu and Guofang Huang, Sister Ye Liu and Brother-in-law Huahua Nie.

Acknowledgement

My PhD dissertation would be impossible without the support from many great people around me. I would like to deliver my sincerest thanks to all of them.

First and foremost, I thank my advisor Dr. H. Howie Huang for the guidance, inspiration and support that he gave me during my PhD study. It was him that opens the door of research to me. During my six-year graduate odyssey, he helped me from countless aspects, e.g., establishing a research problem, writing a paper, giving a presentation, collaborating with other researchers, and mentoring students and many many more. What I remember most is his encouragement when my first graph algorithm paper got rejected by Eurosys '15. There are so many exceptional supports I have fortunately received from him. Without those support, I would never be possible to finish or even pursue my PhD study. He sets up a model of not only an independent researcher but also a respectful friend for me to follow. I sincerely thank him.

I also would like to thank my dissertation committee members, Dr. Ahmed Louri, Dr. Guru Venkataramani, Dr. Milos Doroslovacki and Dr. Timothy Wood. Their excellent suggestions on my research proposal encouraged me to continue my research and helped me improve my dissertation. I cannot finish my dissertation without their guidance.

I am very fortunate to work with great collaborators as well as wonderful friends surrounding me. I express my highest appreciation to my lab mates, Lei Cui, Ahsen Uppal, Pradeep Kumar, Yang Hu, Yuede Ji, and Bibek Bhattarai. I also would like to thank Dr. Ron Chi-Lung Chiang, Dr. Xin Xu, Dr. Juzi Zhao, Dr. Jie Chen, Dr. Yu Xiang and Fan Yao for their support, help and guidance during my PhD journey. I thank Dr. Suresh Subramaniam, Dr. Tian Lan, Dr. Rajat Mittal, Dr. Jung-Hee Seo, Dr. Chen Zeng, and Dr. Yunjie Zhao for their great collaborations.

During my graduate studies, I also had a chance to interact with internationally recognized researchers and receive valuable inputs from them. I am thankful to Dr. Da Zheng (Johns Hopkins University) for his help of my Graphene work (FAST '17) , Dr. Duane Merrill (Nvidia Research) for Enterprise (SC '15) and SIMD-X work, Dr. Yangzihao Wang (UC Davis) and Dr. John D. Owens (UC Davis) for my SIMD-X work and Andrew Gallo for my intern at Division of IT at GWU, and Dr. Cheng-Hong Li for my intern at NEC Laboratories America Inc.

Last but not the least, I want to particularly thank my dearest wife Ni Yang. She stands by my side regardless of my ups and downs. Her persistent encouragement, support and suggestions enable me to pursue my PhD study, finish my dissertation research, and seek future career dreams. Finally, I am grateful to my parents, sister and brother-in-law for always motivating me to pursue my dreams and being there whenever I needed them. Without them, I would not have come this far.

This work is supported in part by the National Science Foundation, and NVIDIA Academic Partnership Award.

Abstract

Novel Techniques for Graph Algorithm Acceleration

The concept of graph has been around since Euler brought up the Seven Bridges of Königsberg problem in 1736. Recent years have seen graph computing regains its momentum because of many emerging graph relevant applications, e.g., World-Wide-Web (WWW) networks, social and computer networks, metabolic interactions and chemical compound design graphs. *This dissertation strives to provide graph computing systems which are able to quickly compute very large graph datasets with relatively low cost and expose easy programming interface to programmers.*

The first part of this dissertation (Chapter 2) introduces the Graphics Processing Units (GPUs) accelerated graph traversal which consists of two projects – Enterprise [1] and iBFS [2]. Particularly, Enterprise is the first work that achieves atomic operation free Breadth-First Search (BFS) on GPUs and iBFS is the first to conduct multiple traversals together on GPUs. Both projects achieve orders of magnitude speedup over state-of-the-art.

Chapter 3 introduces SIMD-X [3], a graph framework that supports a variety of graph algorithms on GPUs. SIMD-X not only provides a simple Active-Compute-Combine (ACC) programming model for end users to express graph algorithms on Single Instruction Multiple Data (SIMD) GPUs, but also creates opportunities for system-level optimizations. Together, SIMD-X allows programmers to develop a typical graph algorithm with less than 100 Lines Of Code (LOCs) and achieve an order of magnitude speedup over Gunrock [4].

Chapter 4 describes Graphene [5] which can tackle trillion-edge (10^{12}) graphs on a single machine with an array of Solid State Drives (SSDs). To enhance the bandwidth utilization of such an array of SSDs, we introduce a bitmap based IO request management component that improves bandwidth efficiency by 4 - 8$\times$ and a row-column 2D graph partition approach to balance the graph data access across the disks. Notably, Graphene achieves comparable performance to in-memory systems, e.g., Galois [6] with merely 10% of memory consumption.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A graph is a powerful structure that can represent relationships between human beings (social network) [11, 12, 13], locations (road map) [14, 15, 16], computers (Internet) [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27], software source code (control and data flow graphs) [28, 29], and biological components (metabolic interactions) [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]. Graph algorithms which analyze graphs and extract valuable insights are of paramount value to our society. However, graph algorithms typically present irregular characteristics, such as workload imbalance and random memory access patterns, which prevent graph algorithms from achieving high-performance. Even worse, with real-world graphs keep growing, e.g., Facebook contains more than 2 billion active users and trillions of edges [12], developing graph analytical systems that can tackle big graphs within a reasonable time envelope is crucial.

The first question we ask is – *"What real-world applications (software) can benefit from a better graph computing system?"* We have kept asking this question during my dissertation study, but very few articles answered with adequate details. This dissertation surveys a broad domain of literatures and presents eight examples of real-world software that can receive immediate benefit from faster graph algorithms, as shown in Table 1.1. Breadth-first search (BFS) is a building block for several graph algorithms, such as centrality computing and vertex relabeling (Reverse Cuthill-Mckee (RCM)) for graph bandwidths reduction. This dissertation finds that BFS is used

Algorithm	Utilization	Software
Breadth-first search (BFS)	P2P file sharing	Gnutella [40]
k-Core	Graph visualization	LaNet-vi [41]
Multi-source BFS	Centrality computing	CentiLib [42]
PageRank	Webpage search	Google search [43]
Single-source shortest path (SSSP)	Navigation	Google map [44]
Sparse matrix vector multiplication (SpMV)	Computational fluid dynamics (CFD)	simFlow [45]
Graph coloring	Register allocation	LLVM [46]
Triangle completing	Friend recommendation	Facebook [47]

Table 1.1: Graph algorithms and software that use them.

by Peer-to-Peer (P2P) file sharing, Gnutella [40], to target sharing neighbors. k-Core decomposition divides a graph into layers (cores) resembling the process of zooming in/out of a network [48]. LaNet-vi [41] is such a type of software. Multi-source BFS [49, 2] algorithm is widely used by centrality computation [2, 49]. CentiLib [42] is the existing software that can benefit from faster centrality computation. Google uses PageRank and single-source shortest path (SSSP) to search the webpages of our interest and finds the fastest path between the source and destination. During my dissertation study [32, 33, 34], We use the sparse matrix vector multiplication (SpMV) algorithm to solve the Navier Stokes equation for cardiac simulation [34]. The popular simFlow [45] can also benefit from our fast SpMV implementation. The graph coloring algorithm is used by Low Level Virtual Machine (LLVM) [46] for register allocations. In social communities, such as Facebook, the triangle completing algorithm [50] is a typical algorithm used for friend recommendation.

Although there exist a certain number of researches about graph computing, they mainly center around optimizing graph algorithms on traditional hardware, such as a Central Processing Unit (CPU) and Hard Disk Drive (HDD). In contrast, the projects proposed and implemented in this dissertation enable much faster graph computing via unleashing the unprecedented power of a variety of new hardware platforms, such as Graphics Processing Unit (GPU) and Solid-State Drive (SSD). The remainder of this chapter is organized as follows.

1. Section 1.1 introduces the prevailing storage format of graphs and the algorithms we accelerate in the follow up sections. For each algorithm, Section 1.1 presents

its essential data structure, operations and control-flows.

2. Section 1.2 discusses the features of contemporary GPU and SSD, as well as comparing their hardware specifications to conventional processors and storage devices, i.e., CPU and HDD, respectively.

3. Section 1.3 describes the challenges of mapping graph algorithms (Section 1.1) on the hardware we discussed in Section 1.2. Specifically, we will discuss graph algorithm's irregular memory access, workload imbalance and hard programmability of mapping graph algorithms on GPUs.

4. Section 1.4 summarizes a variety of novel optimizations from four projects of my dissertation study. These techniques are proven to be beneficial for various graph algorithms on GPUs, as well as SSD-based external memory graph systems.

5. The organization of this dissertation is presented in Section 1.5.

1.1 Graph Algorithms

This section discusses three popular graph storage formats and seven graph algorithms, both of which are essential for graph computing. Particularly, this section unveils the goal (output) of each algorithm along with the control flows and data structures that are necessary to fulfill such a goal on the discussed formats. In general, **a graph algorithm accesses graph data and updates its algorithmic metadata which is the states of vertices or edges in an iterative manner**. We thus will present what the metadata is, how an algorithm accesses both graph data and metadata. Note, the algorithms are presented in an alphabetical order.

1.1.1 Graph Format

Simply put, a graph $G = (V, E, W)$, where V, E and W are the sets of vertices, edges, and edge weights. Figure 1.1(a) plots a simple graph which contains six vertices and

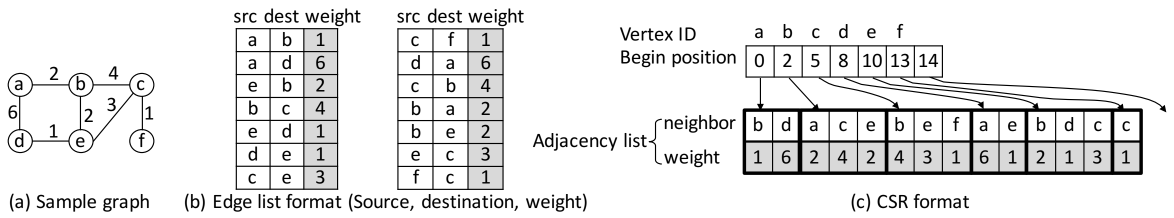


Figure 1.1: (a) An example graph with its storage representation in (b) edge list format and (c) CSR format, respectively.

fourteen directed edges. Note in an undirected graph like Figure 1.1(a), we regard each undirected edge as two directed ones. Typically, graphs are stored in edge list format that is an array of {source, destination, weight}, as shown in Figure 1.1(b). Recent projects introduce the *compressed sparse row* (CSR) format [1] for better access pattern and lower space consumption. Particularly, for the same graph in Figure 1.1(a), CSR sorts all edges by their source vertex, and uses a begin poisition and an adjacency list to represent the graph. In this format, adjacency list stores the destination vertex and the edge weight of each edge. Begin position specifies the range of destination vertices (from adjacency list) that connect to the same source vertex. For instance, vertex a's neighbor vertices fall in the range of 0 to 2. Comparing to edge list format which consumes $3{\cdot}|E|$ space, CSR only consumes $2{\cdot}|E| + |V|$ space. Beyond that, accessing all vertices that connect to the same source vertex experiences sequential consecutive memory access pattern in CSR but edge list format would experience random memory access, e.g., accessing all neighboring vertices of vertex b in Figure 1.1(b).

Complementary to CSR format, there exists a Compressed Sparse Column (CSC) format which sorts edges by destination vertex and only stores the source neighbors of each destination vertex in adjacency list. As a result, CSC format introduces good locality for algorithms that prefer accessing source vertices of each destination vertex together, such as PageRank. As we will discuss shortly, these two formats are often used together for further performance optimizations which is also adopted by this dissertation.

There also exist active works of researches that develop new graph storage formats for better space saving and performance. For instance, G-Store [51, 52] uses a bitwise matrix to store the graph, which benefits dense graph computing tremendously. WebGraph [53] assumes the neighbor list of each vertex contains continuous vertices, which is very common in webpage graphs. As a consequence, WebGraph uses deduplication to reduce the szie of the neighbor list for each vertex thus decreases graph size. [54] also develops graph compression techniques to enable big graph computing in shared memory system.

1.1.2 Graph Algorithms

Breadth-First Search (BFS) [1] traverses a graph level by level. At each level, it loads all neighbors that connect to vertices visited in preceding level (frontier), checks the statuses of those neighbors, and subsequently marks the statuses of those unvisited neighbors as current level. Synchronization is needed at the end of each level. The essential data structures in BFS is status array and frontier queue. Status array, which is indexed by vertex ID with size of |V|, indicates which level this vertex is first time visited. Frontier queue stores frontiers. During the entire traversal process, BFS typically experiences lighter workload at the beginning and end of the computation, while higher workloads exist in the middle levels.

iBFS [2], also called multi-source BFS, is first defined in our iBFS project from this dissertation. This algorithm executes multiple BFS instances on the same graph from different sources simultaneously. Traditionally, this algorithm simply executes all concurrent BFS instances independently, that is, each BFS instance maintains its own metadata and conducts computations separately but share the same graph data. We find this algorithm has a wide range of applications. Particularly, depending on the value of i, iBFS actually becomes a number of different problems. Formally, in a graph with $|V|$ vertices, iBFS is:

- single source shortest path (SSSP) *if* $i = 1$ [55];
- multi-source shortest path (MSSP) *if* $i \in (1, |V|)$[56, 57];
- all-pairs shortest path (APSP) *if* $i = |V|$ [58, 59].

Sparse Matrix Vector Multiplication (SpMV) [33] conducts dot-product between each row of a sparse matrix and a dense vector and stores the output in a result vector. In this algorithm, the sparse matrix and dense vector serve as the roles of graph data and metadata, respectively. Unlike BFS, SpMV is completed in a single iteration thus requires no synchronization. In this dissertation, a number of projects [32, 33, 34] rely on this algorithm for fast CFD simulations.

***k*-Core Decomposition**, which is widely used to study random graph evolution and graph visualization [5, 48], iteratively deletes the vertices whose degree is less than k until all remaining vertices possess more than k neighbors. It is worth pointing out that this algorithm cannot be finished in one iteration because vertices just deleted will affect the degrees of remaining vertices. In this algorithm, the degree array serves as the metadata. Across all iterations, this algorithm experiences large volume of tasks at initial iterations.

PageRank [43] updates the rank value of one vertex based on the contribution of all in-neighbors iteratively till the rank values of all vertices converges. In this algorithm, the rank array of all vertices serves as the metadata. Note this algorithm needs two rank arrays because we need to use the old rank value to generate new ones. Synchronization is necessary at the end of each iteration. Because the contributions of all in neighbors are summarized to each destination vertex, this algorithm prefers graphs in CSC format.

Single-Source Shortest Path (SSSP) computes the shortest path between source vertex and the remaining vertices of the graph. Albeit similar to BFS as traversal algorithm, SSSP is more challenging mainly for the order of computing these active vertices is strictly restricted, that is, the vertex with the shortest distance should be computed first. To improve the parallelism, we adopt the delta-step [60] algorithm, which allows us to compute the vertices whose distances are relatively shorter together. This algorithm is closely related to BFS algorithm except it considers edge weight and can be asynchronous.

Weakly Connected Component (WCC) is a special type of subgraph whose vertices are connected to each other. For directed graphs, a strongly connected component exists if a directed path can be found between all pairs of vertices in the subgraph [61]. In contrast, a WCC exists if such a path can be found regardless of the edge direction. We implement the hybrid WCC detection algorithm presented in [62], that is, it uses BFS to detect the largest WCC then uses label propagation to compute remaining smaller WCCs. In this algorithm, the label array serves as the metadata. Synchronization is needed between BFS and label propagation.

1.2 Computing Hardware

This dissertation mainly takes advantages of two types of hardware for graph computing acceleration – GPU and SSD. This section covers the hardware specifications of GPU which we use in Enterprise [1], iBFS [2] and SIMD-X [3]. For SSD, which is used in Graphene [5], we compare its bandwidth and latency against traditional Hard-Drive Disks (HDDs).

1.2.1 General-Purpose GPUs

This section mainly explains GPU hardware, using NVIDIA Kepler K40 as an example [63]. The K40 consists of 15 Streaming Processors (SMX), each of which has 192 single-precision CUDA cores and 64 double-precision units. Each GPU **thread** runs on one CUDA core and an SMX schedules the threads in a group of 32 that is called a **Warp**. Figure 1.2 presents an overview of GPU architecture.

An SMX can support up to 64 warps. All the threads in a warp are executed in the single-instruction, multiple-thread fashion. But if the threads in a warp have different control paths, the warp executes all the taken branches sequentially and disables each individual thread that is not on the taken path. This so called branch divergence problem, if exists, could largely reduce SMX utilization.

Each SMX features four **Warp Schedulers** which select four warps in round-robin and issue the instructions from those that are ready for execution. The warps that are not ready due to long latency data accesses are skipped. By oversubscribing threads in each SMX, data access can be overlapped with execution.

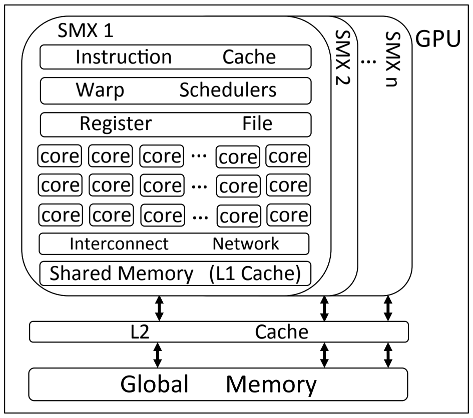


Figure 1.2: A simplified view of GPU architecture.

Cooperative Thread Array (CTA), thread block, consists of multiple warps, typically 1 to 64, which can be used to run a large number of threads. And the set of all the CTAs on a GPU is referred to as a **Grid**. The number of CTAs and the number of threads in each CTA are configurable. Each thread in a CTA has a unique Thread ID and each CTA has its own CTA ID. With these built-in variables, one is able to identify each thread in a grid and schedule different threads to work on different data.

A Kernel is defined as any function that runs on GPUs. Typically, one kernel can use different **parallel granularity** (i.e., a thread, warp, CTA, or grid) by employing a certain quantity of threads. Kepler introduces Hyper-Q to support concurrent kernel execution, in other words, when several kernels are executed on the same GPU, Hyper-Q is able to schedule them to run on different SMXs in parallel to fully utilize all GPU resources.

GPU Memory Hierarchy. Each SMX has a large number of **registers**, e.g., 65,536 for each K40 SMX. Each thread can use up to 255 registers and perform four register access for each clock cycle. In addition, each SMX provides software configurable **shared memory** (L1 cache) for intra-warp and intra-CTA data communication. Each K40 SMX has 64 KB of shared memory. Different from the L1 cache on CPUs, one can allocate 16, 32, or 48 KB of the shared memory at the program runtime. Once loaded, the data in the shared memory is readable and writable to all the threads in one CTA.

Memory	CPU		GPU		
	Size	Latency	Size	Latency	BFS Data Structures
Register	12	1	65,536	-	Status Array
L1 cache	64KB	4	64KB	-	Hub Vertex Cache
L2 cache	256KB	10	1.5MB	-	-
L3 cache	24MB	40	-	-	-
DRAM	up to 2TB	55	12GB	200 - 400	Status Array, Frontier Queue, Adjacency List

Table 1.2: CPU (Xeon E7-4860) vs. GPU (K40) memory: size and access latency (in CPU and GPU cycles) [8, 9]

GPU also has the **L2 cache** and **global memory** that are shared by all SMXs. The K40 has 1.5 MB L2 and 12 GB global memory. Each global memory access is replied with a data block that contains 32, 64 or 128 bytes based on the type. If a warp of threads happen to access the data in the same block, only one hardware access transaction is performed. By coalescing global memory accesses into fewer transactions in this way, K40 is able to achieve close to 300GB/s DRAM bandwidth.

Table 1.2 summarizes the CPU and GPU memory hierarchies. Note that K40 has no L3 cache. We cannot find official latency numbers for register and shared memory, but our tests show that they are at least an order of magnitude faster than the global memory. In this work, we leverage the GPU support of concurrent kernels and different parallel granularity to match dynamic BFS workloads, and utilizes different GPU memory for various BFS data structures, e.g., using shared memory for hub vertices.

GPU Hardware Performance Counters. GPUs today can support more than 100 hardware metrics [64]. In this work, we aim to understand the kernel performance, GPU I/O throughput, and energy efficiency of our system, including the timeline of different kernels, utilization of memory load/store function unit (*ldst_fu_utilization*), percentage of stalls caused by data requests (*stall_data_request*), global memory load transactions (*gld_transactions*), IPC and power. We use two NVIDIA tools, i.e., *nvprof* and *nvvp*.

Device	Read		Power consumption (mW)	Price/1 TB (USD)
	Sequential (MB/s)	Random (IOPS)		
HDD	146	169	817	60
SSD	550	33,000	30	250

Table 1.3: SSD vs. HDD [10].

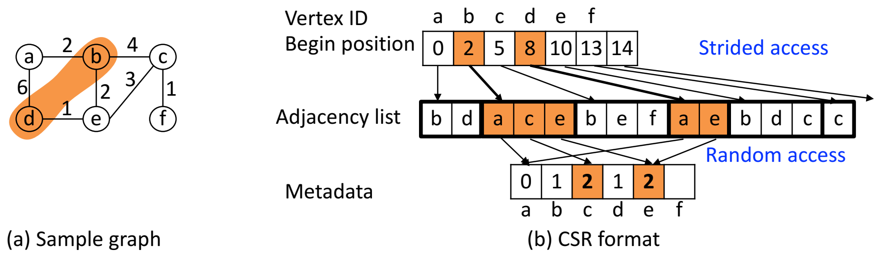


Figure 1.3: Assuming BFS traverses the sample graph and vertex {b, d} are active.

1.2.2 SSD

This section compares the read throughput, power consumption and price differences between SSD and HDD, particularly, Z400s SSD and 5400 HDD. As shown in Table 1.3, SSD provides 3.8× times higher sequential throughput than HDDs with also 4.2× times the price of HDDs. However, it is important to notice that SSDs provide hundreds of times higher random read throughput than HDDs which is crucial for graph computing that mainly presents random read patterns. In addition to performance, SSDs also consumes 27× less power than HDDs, which is very important for green graph computing [65, 66, 67, 68].

1.3 Challenges

GPU and SSD provide exceptional computing and data delivering performance which is attractive for deploying graph analytics. However, exploiting the potentials of these two devices faces non-trivial challenges, which are irregular memory access (Section 1.3.1), workload imbalance (Section 1.3.2), and difficult programmability (Section 1.3.3).

1.3.1 Irregular Memory Access

A graph algorithm typically experiences random access for both graph data and algorithmic metadata. Random accesses to graph data results from pointer-chasing nature of graph algorithms, that is, each iteration, graph algorithms often access a subgraph that is decided by previous iteration during runtime. Algorithmic metadata typically experiences even more random memory access patterns because of indirect memory access. Both of them can be explained by the BFS example in Figure 1.3. We assume vertices {b, d} are visited in the preceding level. BFS needs to first access the begin position of vertex *b* and *d*. Afterwards, BFS loads the neighbors of these two vertices which are {a, c, e} and {a, e}. This is random graph data access. Then BFS checks the status (metadata) of these neighbors by their neighbor IDs and marks those unvisited neighbors as depth of 2, that is, accessing metadata[0], metadata[2] and metadata[4] for the statuses of {a, c, e} and metadata[0] and metadata[4] for {a, e}. This is random memory access for metadata.

In this dissertation, we design a variety of techniques to alleviate the expensive random memory access for graph data and metadata. For instance, Enterprise [1] sorts the frontiers before accessing their neighbors so that nearby active vertices access their neighbors consecutively. iBFS [2] combines the metadata access of multiple BFS instances so that accessing the metadata of the same neighbor vertex fits in the same cache line.

1.3.2 Highly Skewed Workload Distribution

This problem stems from the fact that in real-world graphs, vertices, typically, have drastically different degrees – number of neighbors, which is called power-law degree distribution [69]. Figure 1.4 presents the degree distribution of a popular dataset – Twitter graph which contains 53 million vertices and 2 billion edges. In this graph, there are more than ten million vertices which have the degree of only one while one vertex with nearly one million neighbors. The degree differences between these two extreme cases is 10^6.

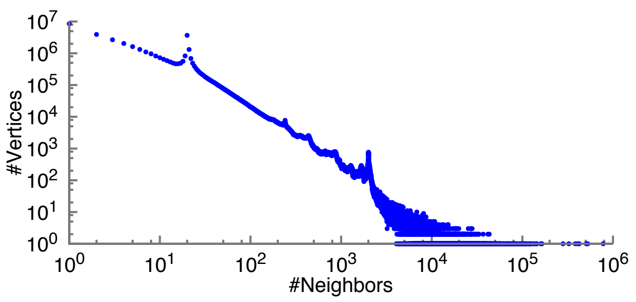


Figure 1.4: Degree distribution of Twitter Graph [7] which contains 53 million vertices and 2 billion edges.

This degree distribution is detrimental for graph computing because vertex degree is closely related to the amount of workload that is binding to this vertex. For instance, in BFS, iBFS, PageRank, SSSP, k-Core, WCC and SpMV, the amount of neighbors is exactly the number of workloads, i.e., workload $\propto$ degree. Therefore, balancing workload across threads, machines or SSDs is crucial.

It is important to note that GPU and CPU addresses such a problem differently. Particularly, for GPU-based system which contains abundant amount of threads, we vary the amount of threads assigned to active vertices in order to balance the workload across threads. For CPU-based system, which contains merely tens of threads, we rely on graph partitioning optimization to balance the workload across SSDs and CPU threads.

1.3.3 Difficult Programmability

Programming graph algorithms is difficult for several of reasons. First, the computing process contains several indirections. Using BFS as an example, we need to access the frontiers from frontier queue and using the dequeued value to index the begin position. With the begin position, we have to access the adjacency list. Further with the adjacency list, we need to access the status array. Second, we need to manage several data structures. In BFS, we need to manage the adjacency list, begin position, status array and frontier queue. Third, graph computation contains a lot of optimization techniques such as push-pull model [70, 71], delta step model [60, 72], and hybrid computation [73].

Graph system	LOC of Programmers	GPU vs. CPU
B40C [74]	5,000	GPU
Enterprise [1]	3,000	GPU
Gunrock [75]	600	GPU
Galois [6]	500	CPU
FlashGraph [76]	700	CPU

Table 1.4: Summary of related work for BFS programming difficulties.

Mapping graph algorithms to GPUs and SSDs introduces two more difficulties, i.e., GPU programming and IO stack management. For GPU, a programmer needs to explicitly manage different granularity of threads and hierarchies of memory, as well as data flows between different data structures in a lock-free and high thread utilization manner. To further exacerbate the situation, GPU also does not provide global synchronization techniques, which is essential for the majority of the graph algorithms. For SSD, managing an array of SSDs in order to provide high-throughput graph computing is also challenging. Users need to understand the entire Linux IO stack, which does not contain any GLIBC error handling mechanisms. Beyond that, users also have to understand file and memory alignment for the correctness and most efficient IO utilization.

Table 1.4, from lines of code (LOC) perspective, approximates the programming difficulties of developing BFS on GPU (B40C, Enterprise), GPU-based graph system (Gunrock), in memory system (Galois) and external memory system (FlashGraph). For highly optimized GPU based graph system, like B40C and Enterprise, it takes several thousands of LOCs to fulfill the functionalities of BFS. Even an usability optimized system – Gunrock takes 600 LOCs to implement a BFS algorithm. For external memory system, it takes, again, more than 500 LOCs to implement a BFS algorithm.

In this dissertation, we introduces SIMD-X [3] for GPU based graph computing. With SIMD architecture, users enjoy very simple data parallel programming abstraction. For external memory graph computing [5], we use "Think Like an IO request" API to completely hide the IO complexities from programmers. Both abstractions minimize the programmers' involvement for the system optimizations

1.4 Dissertation Contributions

This dissertation encompasses five projects which fall in the topics about graph traversal, graph systems and applying graph algorithms to biomedical research. The application related project is discussed in the beginning of this dissertation. This section summarizes the remaining four projects into three aspects, GPU-based graph traversal (Section 1.4.1), GPU-accelerated graph computing (Section 1.4.2) and SSD-based graph computing (Section 1.4.3).

1.4.1 Graph Traversal on GPUs

Enterprise [1] and iBFS [2] are the two projects related to graph traversal. Enterprise achieves high-performance graph traversal through combining three techniques to remove potential performance bottlenecks: (1) *streamlined GPU threads scheduling* through constructing a frontier queue without contention from concurrent threads, yet containing no duplicated frontiers and optimized for both top-down and bottom-up BFS. (2) *GPU workload balancing* that classifies the frontiers based on different out-degrees to utilize the full spectrum of GPU parallel granularity, which significantly increases thread-level parallelism; and (3) *GPU based BFS direction optimization* quantifies the effect of hub vertices on direction-switching and selectively caches a small set of critical hub vertices in the limited GPU shared memory to reduce expensive random data accesses. We have evaluated Enterprise on a large variety of graphs with different GPU devices. Enterprise achieves up to 76 billion traversed edges per second (TEPS) on a single NVIDIA Kepler K40, and up to 122 billion TEPS on two GPUs that ranks No. 45 in the Graph 500 on November 2014. Enterprise is also very energy-efficient as No. 1 in the GreenGraph 500 (small data category), delivering 446 million TEPS per watt.

In iBFS, we focus on a special class of graph traversal algorithm - concurrent BFS - where multiple breadth-first traversals are performed simultaneously on the same graph. iBFS consists of three novel designs. First, iBFS develops a single GPU kernel for *joint traversal* of concurrent BFS to take advantage of shared frontiers across different instances. Second, *outdegree-based GroupBy rules* enables iBFS to selectively run a group of BFS instances which further maximizes the frontier sharing within such a group. Third, iBFS brings additional performance benefit by utilizing highly optimized *bitwise operations* on GPUs, which allows a single GPU thread to inspect a vertex for concurrent BFS instances. The evaluation on a wide spectrum of graph benchmarks shows that iBFS on one GPU runs up to 30× faster than executing BFS instances sequentially, and on 112 GPUs achieves near linear speedup with the maximum performance of 57,267 billion traversed edges per second (TEPS). Chapter 2 discusses this project further.

1.4.2 SIMD-X: Programming and Processing of Graph Algorithms on GPUs

With high computation power and memory bandwidth, graphics processing units (GPUs) lend themselves to accelerate data-intensive applications, especially when they fit the *single instruction multiple data* (SIMD) model. However, graph algorithms such as breadth-first search and k-core, often fails to take full advantage of GPUs, due to irregularity in memory access and control flow. To address this challenge, we have developed SIMD-X, for programming and processing of *single instruction multiple, complex, data* on GPUs. Specifically, the new *Active-Compute-Combine* (ACC) model not only provides ease of programming to programmers, but more importantly creates opportunities for system-level optimization. To this end, SIMD-X utilizes *just-in-time task management* which at runtime filters out inactive vertices and dynamically maps parallel tasks of graph computing to GPUs. In addition, SIMD-X leverages *push-pull based kernel fusion* that with the help of a new deadlock-free global barrier, reduces a large number of computation kernels to very few. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving up to an order of magnitude speedup compared to the state-of-the-art. Chapter 3 discusses this project further.

1.4.3 Graphene: Fine-Grained IO Management for Graph Computing

As graphs continue to grow, external memory graph processing systems serve as a promising alternative to in-memory solutions for low cost and high scalability. Unfortunately, not only does this approach require considerable efforts in programming and IO management, but its performance also lags behind, in some cases by an order of magnitude. In this work, we strive to achieve an ambitious goal of achieving ease of programming and high IO performance (as in-memory processing) while maintaining graph data on disks (as external memory processing). To this end, we have designed and developed *Graphene* that consists of four new techniques: an IO request centric programming model, bitmap based asynchronous IO, direct hugepage support, and data and workload balancing. The evaluation shows that Graphene can not only run several times faster than several external-memory processing systems, but also performs comparably with in-memory processing on large graphs. This project is further discussed in Chapter 4.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 studies *Enterprise* – GPU acceleration of BFS and *iBFS* – the novel designs of concurrent traversals. Chapter 3 describes *SIMD-X*, and Chapter 4 discusses the Graphene system for external memory big graph processing. Finally in Chapter 5, we conclude this dissertation.

Chapter 2

Graph Traversal on GPUs

2.1 Introduction

Breadth-First Search (BFS) algorithm serves as a building block for many analytics workloads, e.g., single source shortest path, betweenness centrality [77, 78, 79, 80] and closeness centrality [81, 82]. Notably, the Graph 500 benchmark uses BFS on power-law graph to evaluate high-performance hardware architectures and software systems that are designed to run data-intensive applications [83]. In this work, we are particularly interested in accelerating BFS traversal on power-law graphs, which can be found in a wide spectrum of applications, e.g., biomedical cells [84], WWW [85, 86] and social network [87, 7, 88, 89, 90].

The traditional (top-down) BFS algorithm starts at the root of the graph and inspects the status of all of its adjacent (or neighboring) vertices. If any adjacent vertex is unvisited, the algorithm will identify it as a frontier, put it into a queue that we refer to as the *frontier queue* in this paper, and subsequently mark it visited. As the result of the inspection of the current level, the frontier queue consists of all the vertices that have just been visited and will be used for expansion at the next level. To do so, BFS iteratively selects each vertex in the frontier queue, inspects its adjacent vertices, and marks this vertex visited. The process of expansion and inspection is repeated level by level till no vertex in this graph remains unvisited. For recently proposed bottom-up BFS [70], the workflow is similar with different vertices identified as frontiers. Clearly, the frontier queue is at the heart of the BFS algorithm - at each level BFS starts with the frontier queue prepared by the inspection of the preceding level and ends with a new frontier queue that will be used for the expansion of next level.

Graphics Processing Unit (GPU) provides not only massive parallelism (in 100Ks threads) but also fast I/O (with 100s GB/s memory bandwidth), which makes it an excellent hardware platform for running the BFS algorithm. Unfortunately, although recent attempts [55, 74, 91, 92] have made remarkable progress, unleashing the full power of GPUs to achieve high-performance BFS remains extremely challenging. In this paper, we advocate that a high-performance BFS system shall carefully match the hardware aspects of GPUs through efficient management of numerous GPU streaming processors and unique memory hierarchy.

In this chapter, we present Enterprise[1], a new GPU-based BFS system that tailors the BFS execution flow and data access pattern to take full advantage of high thread count and massive memory bandwidth of GPUs. Enterprise achieves up to 76 billion traversed edges per second (TEPS) on a single NVIDIA Kepler K40, and up to 122 billion TEPS and 446 million TEPS per watt on two GPUs, which ranks No. 45 and No. 1 in the Graph 500 and GreenGraph 500 small data category in November 2014, respectively. This is achieved through the design of three novel techniques:

First, **streamlined GPU threads scheduling** is achieved through efficient frontier queue generation of two distinct steps: the scan of the status array of the graph at the current level, followed by prefix sum based frontier queue generation. When enqueueing a frontier, atomic operations are needed to ensure the uniqueness of each frontier vertex in the queue, however, for GPUs such operations can lead to expensive overhead among a large quantity of GPU threads. By breaking the queue generation into two steps, Enterprise is able to not only eliminate the need of thread synchronization by updating and accessing the status array in parallel, but also remove duplicated frontiers from the queue that avoids potentially useless work down the road. This is further combined with memory optimization to accelerate both top-down and

[1]Enterprise is the name of the first space shuttle built for NASA on 1976.

bottom-up BFS. The evaluation shows that although it may take a small amount of time for queue generation, our GPU threads scheduling can speed up the overall BFS runtime by 37.5×.

Second, **GPU workload balancing via frontier classification**. To mitigate inter-thread workload imbalance, Enterprise classifies the frontiers based on the out-degrees (the number of edges to adjacent vertices) into a number of queues, and assigns a different number of threads to work on each queue. Specifically, Enterprise creates four different frontier queues corresponding to Thread, Warp, Cooperative Thread Array (CTA), and Grid [9]. For example, Enterprise may assign a single thread for the frontiers whose out-degree is less than 32 and a warp for those less than 256. Enterprise may even assign all threads on one GPU to a frontier in the case of extremely high out-degrees (e.g., 10^6). Prior work utilizes a fixed number of threads (typically 32 or 256), where static assignments often result in skewed workload among threads [55, 74, 73, 93]. The frontier classification greatly mitigates this imbalance, leading to additional speedup of 1.6× to 4.1× on top of the proposed GPU threads scheduling technique.

Third, **GPU-aware direction optimization** is developed in Enterprise to run bottom-up BFS efficiently on GPUs. Specifically, we propose a new parameter that uses the ratio of hub vertices in the frontier queue to determine the one-time switch from top-down to bottom-up on GPUs. This parameter is shown to be stable across different graphs, removing the need for parameter tuning as in the prior approach [70]. More importantly, Enterprise selectively caches the hub vertices in GPU shared memory to reduce the expensive random global memory access. Interestingly, this shared memory based cache with a small size of 48 KB, when caching a few thousand of critical hub vertices, can help to reduce up to 95% of global memory transactions in bottom-up BFS.

To the best of our knowledge, Enterprise is the first GPU-based BFS system that not only leverages a variety of GPU thread groups to balance irregular workloads but also employs different GPU memories to mitigate random accesses, both of which are inherent characteristics of graph traversal on power-law graph and especially challenging to optimize on modern GPUs. Enterprise can be utilized to support a number of graph algorithms such as single source shortest path, diameter detection, strongly connected component, and betweenness centrality.

This chapter also accelerates a special class of BFS algorithm - concurrent BFS - where multiple breadth-first traversals are performed simultaneously on the same graph. We refer to our solution to this problem as iBFS that is able to perform i multiple breadth-first traversals in parallel on GPUs, each from a distinct source vertex. Here i is between 1 and $|V|$, i.e., the total number of vertices in the graph.

Depending on the value of i, SIMD-X actually becomes a number of different problems. Formally, in a graph with $|V|$ vertices, SIMD-X is

- single source shortest path (SSSP) *if* $i = 1$ [55];
- multi-source shortest path (MSSP) *if* $i \in (1, |V|)$[56, 57];
- all-pairs shortest path (APSP) *if* $i = |V|$ [58, 59].

Moreover, SIMD-X can be utilized in many other graph algorithms such as betweenness centrality [94, 78] and closeness centrality [81]. For example, one can leverage SIMD-X to construct the index for answering graph reachability queries, that is, whether there exists a path from vertex s to t with the number of edges in-between less than k [95, 96, 97]. We will show in this paper this step can be an order of magnitude faster with iBFS. In all, a wide variety of applications, e.g., network routing [77, 98, 80], network attack detection [99], route planning [100, 101, 102], web crawling [103, 104], etc., can benefit from high-performance SIMD-X.

This dissertation proposes a new approach for running SIMD-X on GPUs that consists of three novel techniques: joint traversal, GroupBy, and bitwise optimization. Prior work has proposed to combine the execution of different BFS instances mostly on multi-core CPUs [49, 82, 105]. The performance improvement, however, is limited. For one, none of early projects has attempted to group the BFS instances to improve the frontier sharing during the traversal. Further, bottom-up BFS provides new additional challenges. For example, while MS-BFS [49] supports bottom-up, it does not provide early termination which SIMD-X leverages for faster traversal. In essence, BFS is a memory-intensive workload that matches well with thousands of lightweight threads provided by GPUs. Prior work such as [55, 74, 91, 92] has shown great success of using GPUs for single-source BFS. To the best of our knowledge, this is the first work that supports concurrent BFS on GPUs.

The first technique of SIMD-X is motivated by the observation that a naive implementation that simply runs multiple BFS instances sequentially or in parallel would not be able to achieve high parallelism on GPUs. To address this challenge, SIMD-X proposes the technique of *joint traversal* to leverage the shared frontiers among different concurrent BFS instances, as they can account for as high as 48.6% of the vertices in some of the graphs we have evaluated. In particular, SIMD-X executes different traversals within a single GPU kernel. That is, all concurrent BFS instances share a joint frontier queue and a joint status array.

Second, to achieve the maximum benefit of joint traversal, SIMD-X shall execute all BFS instances together. However, this is impossible due to limited hardware resources, e.g., global memory size and thread count, on GPUs. Fortunately, we discover that grouping BFS instances can be optimized to ensure a high ratio of frontier sharing among different instances. Guided by a theorem on inter-group sharing ratio, SIMD-X develops the second technique of *GroupBy* that selectively combines BFS instances into optimized batches, which results in 2× speedup of the overall performance.

A typical BFS algorithm starts from the source vertex in a top-down fashion, inspects the frontiers at each level, and switches to bottom-up to avoid inspect too many edges unnecessarily. GroupBy improves iBFS performance in both top-down and bottom-up, although in different fashions. In top-down, higher sharing ratio directly reduces the number of memory access for inspection and expansion. In contrast, through sharing, GroupBy allows bottom-up traversals to complete in approximately the same amount of time, minimizing workload imbalance across multiple BFSes.

Third, SIMD-X would need to inspect a considerable amount of frontiers at multiple levels, in some case, upto 15× more than a single BFS. Although for each individual BFS not every vertex is a frontier at every level, concurrent BFS significantly increases the number of frontiers at each level. While GPUs offer thousands

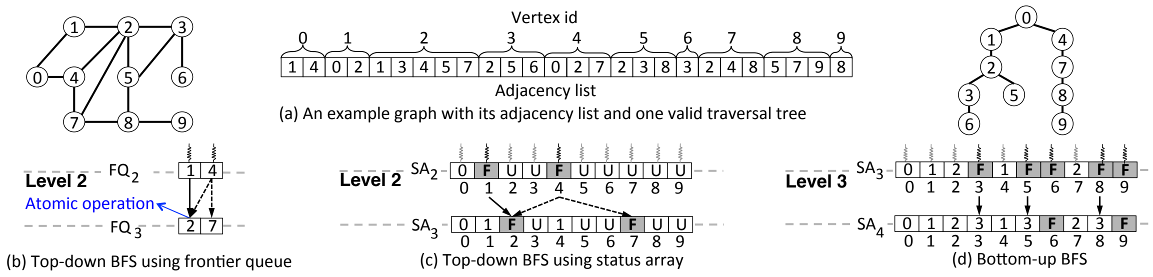

Vertex id
Adjacency list
(a) An example graph with its adjacency list and one valid traversal tree
Level 2
FQ2
Atomic operation
FQ3
(b) Top-down BFS using frontier queue
SA2
SA3
(c) Top-down BFS using status array
Level 3
SA4
(d) Bottom-up BFS

Figure 2.1: (a) An example graph with its adjacency list and one valid BFS traversal tree (there may exist multiple valid BFS trees). We use this example throughout the paper. Top-down BFS using (b) the frontier queue or (c) status array, vs. (d) bottom-up BFS. The numbers in the status array represent at which level the vertex is visited. The labels of F and U represent frontier and unvisited vertex, respectively. In (c) and (d), the gray threads that are assigned to non-frontier vertices would idle with no work.

of hardware threads, traversing millions of the vertices of large graphs in parallel remains challenging. To this end, SIMD-X utilizes a *bitwise status array* that uses one bit to represent the status of the vertex for each BFS instance. This reduces the size of data fetched during inspection, and more importantly through bitwise operations reduces the number of threads needed for inspection, together accelerating the graph traversal by 11×.

The rest of the chapter is organized as follows: Section 2.2 introduces the background on GPUs, BFS, and the graphs used in this paper. Section 2.3 discusses the related work. Section 2.4 presents the challenges of running BFS algorithms on GPUs. Section 2.5 and Section 2.6 describe Enterprise and iBFS, respectively. We further present the overall performance of Enterprise and iBFS in Section 2.7 and Section 2.8, respectively.

2.2 Background

2.2.1 Breadth-First Search

Traditional (top-down) BFS algorithm performs expansion and inspection at each level, that is, from each frontier (last recently visited) vertex v, examining whether an adjacent vertex w is first-time visited. If so, v becomes the parent and w is also enqueued into the frontier queue.

The frontiers can be generated in two ways. In the first approach of Figure 2.1(b), known as atomic operation based frontier generation [106], two threads are dispatched at level 2 to check the adjacency lists of vertices 1 and 4 in the queue FQ_2, and both would like to put vertex 2 into FQ_3. In this case, atomic operations (e.g., atomicCAS in CUDA [9]) are utilized to ensure that FQ_3 has no duplicated frontiers, where whichever thread that finishes first would become the parent of vertex 2. Without atomic operations, vertex 2 would be enqueued twice, resulting in redundant work at level 3.

Since inter-thread synchronization is costly on GPUs, a second approach [92, 91] uses a data structure called the Status Array (SA) to track the status of each vertex in the graph. Status array is basically a byte array indexed by the vertex ID. The status of a vertex can be *unvisited*, *frontier* or *visited* (represented by its BFS level). At every level, a thread will be assigned to each vertex, whereas only those that are working on the frontiers will perform expansion and inspection. Thus, as shown in Figure 2.1(c), while ten threads will be used at level 2, only two will be working on vertices 1 and 4. The advantage of this approach is that atomic operation is no longer needed - both vertices 1 and 4 can be the parent of vertex 2, and the update of the status of vertex 2 can be performed sequentially. Here, unlike the first approach whoever finishes last becomes vertex 2's parent.

Hybrid BFS is initialized with the top-down approach and switches the direction between top-down and bottom-up when the switching parameters satisfy the predefined thresholds. Figure 2.2 presents the workflow of hybrid (direction-optimizing) BFS. Top-down BFS aims to identify the edges that connect the frontiers and unvisited vertices, while bottom-up aims to identify those between the frontiers to visited vertices. This paper formally defines a frontier as:

Definition (Frontier) Let v be a vertex of the graph G. At level i, v becomes a frontier if

- Top-down BFS: v was visited at level $i-1$; or

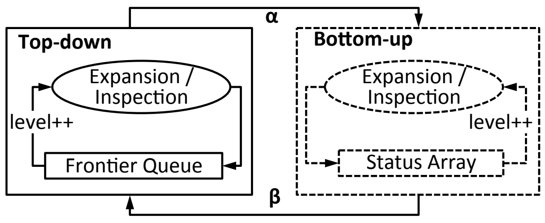


Figure 2.2: Hybrid (direction-optimizing) BFS.

- Bottom-up BFS: v has not been visited between level 0 and $i-1$.

Using the same example in Figure 2.1, at level 2, top-down selects vertices {1, 4} as the frontiers. In comparison, bottom-up uses unvisited vertices {3, 5, 6, 8, 9} as the frontiers at level 3. When bottom-up discovers that vertices {3, 5} connect to a visited vertex 2, they are marked as visited with 2 as the parent. Similarly, vertex 8 is marked as visited with 7 as the parent.

The goal of direction-switching is to reduce a potentially large number of unnecessary edge checks. Hybrid BFS may switch direction twice in the process, i.e., from top-down to bottom-up and from bottom-up to top-down, each of which is associated with a parameter. In Figure 2.2, α is calculated as the ratio of m_u and m_f, where m_u represents the unexplored edge count, and m_f the edges to be checked from the top-down direction; and β is calculated as the ratio of n and n_f, where n represents the number of vertices in the graph and n_f the number of vertices in the frontier queue. Currently the thresholds are heuristically determined.

Switching from bottom-up to top-down is done in the final stages of BFS to avoid the long tail in the graphs, which we find is neither necessary nor beneficial for Enterprise. In this paper, we will show that building an efficient hybrid BFS system will require a number of GPU-aware optimizations, including a stable direction-switching parameter, hub vertex cache, as well as streamlined GPU threads scheduling and workload balancing.

2.2.2 Concurrent BFS

Concurrent BFS executes multiple BFS instances from different source vertices. Using the example in Figure 2.3, four BFS instances start from vertex 0, 3, 6, and 8,

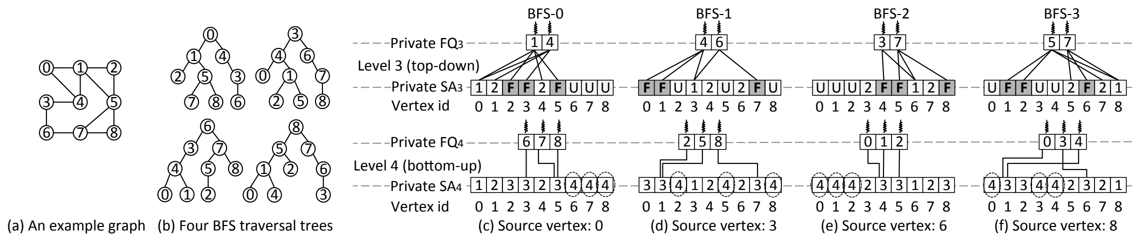


Figure 2.3: (a) An example graph used throughout this paper (b) Four valid BFS traversal trees from different source vertices, i.e., BFS-0, BFS-1, BFS-2, and BFS-3 starting from vertices 0, 3, 6, and 8, respectively. (c), (d), (e) and (f) are two levels traversal of the corresponding four valid trees, where the top half represents top-down traversal and the bottom half bottom-up. In all examples, FQ and SA stand for frontier queue and status array, respectively. The frontier queue stores frontiers while the status array indicates the status of each frontier in the current level where "F", "U" and numbers represent "Frontier", "Unvisited" and its depth (visited), respectively. The dotted circles indicate the updated depth for each frontier.

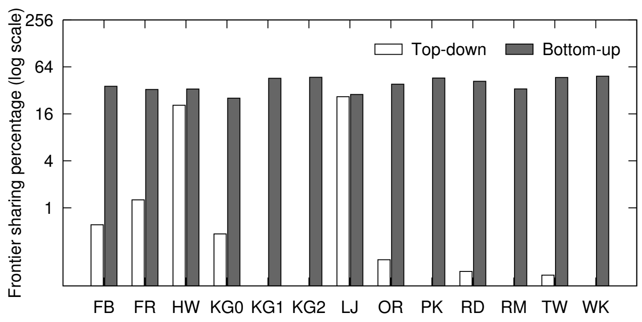


Figure 2.4: Average frontier sharing percentage between two different BFS instances.

respectively. A naive implementation of concurrent BFS will run all BFS instances separately and keep its own private frontier queue and status array. On a GPU device, each individual subroutine is defined as a *Kernel*. Therefore, in the aforementioned example, four kernels will run four BFS instances in parallel from four source vertices. NVIDIA Kepler provides *Hyper-Q* to support concurrent execution of multiple kernels, which dramatically increases the GPU utilization especially when a single kernel cannot fully utilize the GPU [63].

Unfortunately, this naive implementation of concurrent BFS takes approximately the same amount of time as running these BFS instances sequentially, as we will show later in Section 2.8. For example, for all the graphs evaluated in this paper, sequential and naive implementation of concurrent BFS take average 52 ms and 48 ms, respectively, with a difference in traversal rate of 500 million TEPS. The main

reason for such a small benefit is because simply running multiple BFS instances in parallel would overwhelm the GPU, especially at the *direction-switching level* when a BFS goes from top-down to bottom-up. At that moment each individual BFS would require a large number of threads for their workloads. As a result, such a naive implementation may even underperform a sequential execution of all BFS instances.
Opportunity of Frontier Sharing: iBFS aims to address this problem by leveraging the existence of frontiers shared among different BFS instances. Figure 2.4 presents the average percentage of shared frontiers per level between two instances. The graphs used in this paper are presented in Section 2.8. Top-down levels have smaller number of shared frontiers (close to 4% on average) whereas bottom-up levels have much more as high as 48.6%. This is because bottom-up traversals often start from a large number of unvisited vertices (frontiers in this case) and search for their parents. The proposed GroupBy technique can improve the sharing for both directions to 10× and 1.7×, respectively.

Potentially, the shared frontiers can yield three benefits in concurrent BFS: (1). These frontiers need to be enqueued only once into the frontier queue. (2). The neighbors of shared frontiers need to be loaded in-core only once during expansion. (3). Memory accesses to the statuses of those neighbors for different BFSes can be coalesced. It is important to note that each BFS still has to inspect the statuses independently, because not all BFSes will have the same statuses for their neighbors. In other words, shared frontiers do not reduce the overall workload. Nevertheless, this work proposes that shared frontiers can be utilized to offer faster data access and saving in memory usage, both of which are critical on GPUs. This is achieved through a combination of bitwise, joint traversals and GroupBy rules that guide the selection of groups of BFSes for parallel execution.

2.3 Related Work

Our system Enterprise advances the state of the art in the design and implementation of graph traversal. Prior work uses either the frontier queue [106, 74] or status array [92]. Even when using both data structures, existing solutions use them at different directions, e.g., [91, 93] use the status array at the explosion level and the frontier queue method at other levels, and [70] uses the frontier queue for top-down and the status array for bottom-up. Enterprise utilizes both data structures throughout and delivers unprecedented performance on GPUs.

Recently several workload balance techniques have been proposed for GPUs such as task stealing [107, 108] and workload donation [109, 110]. However, this type of technique is often used in a small group of threads, and is extremely challenging to coordinate among thousands of threads as we have in this work. Instead, Enterprise targets the root of BFS workload imbalance and classifies different frontiers to mitigate the problem.

There are a number of projects [111, 112, 113] that leverage hub vertices to reduce the communication overhead, especially for distributed BFS. For example, [111] duplicate the status of hub vertices across all the machines at every level, and [112] and [113] divide hub vertices into multiple partitions and communicate in a tree-based manner. In contrast, Enterprise only enables the hub vertex cache for bottom-up levels when expansion and inspection center around hub vertices. Additionally, as GPU shared memory is limited, Enterprise updates the cache at each level with those who most likely will be visited in the following level.

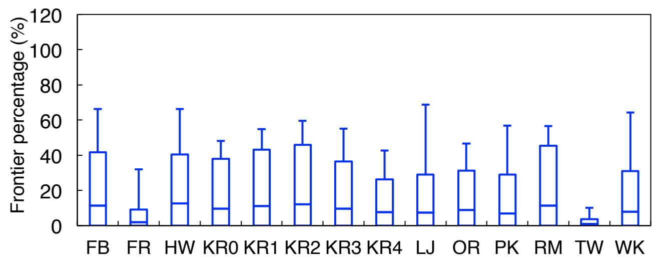
Our iBFS is closely related to CPU-based MS-BFS [49], which runs concurrent BFS by extending a single-threaded BFS, and compared to iBFS, underperforms on large graphs, e.g., it only achieves 10 billion TEPS on a graph with a billion of edges. In contrast, iBFS achieves more than 500 billion TEPS on a graph with billions of edges. Furthermore, the bitwise operation from MS-BFS cannot support early termination due to that it requires to reset the status array at each level, which leads to a much slower performance. Most notably, iBFS introduces a novel GroupBy strategy that improves the frontier sharing ratio dramatically and increases the benefit of concurrent traversal by another $2\times$.

Recent work has demonstrated that GPUs have great potentials in delivering high-performance breadth-first graph traversal [73, 74]. One project SpMM-BC on regularized centrality [82] also extends the GPU-based BFS to concurrent BFS, but it does not support bottom-up BFS. On the other hand, the work [79] executes concurrent BFS to calculate the betweenness centrality [94] of a graph. However, each GPU in this work only executes a single BFS which is similar to the *naive* implementation in our work. In comparison, iBFS runs hundreds of BFS instances with one kernel and achieves on average 22$\times$ speedup compared to executing a single BFS on one GPU. Also, iBFS supports $\mathcal{O}(100)$ GPUs and achieves around 200 billion TEPS for graphs such as LJ and OR, significantly outperforming prior work [82, 79].

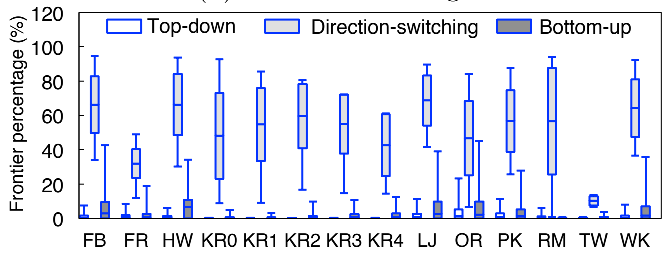
In addition, our work is related to three types of shortest path algorithms [114, 115, 116, 117, 118, 119, 120], namely, Dijkstra, Bellman-Ford and Floyd-Warshall. The first two algorithms focus on SSSP while the later APSP. Specifically, Dijkstra [121] applies to weighted graph where weight must be positive, with the complexity of $\mathcal{O}(|E| + |V| \log |V|)$. Bellman-Ford [122] extends Dijkstra by allowing negative weighted edges but non-negative cycles. In contrast, our iBFS applies to all types of shortest path problems on a unweighted graph with the time complexity of $\mathcal{O}(i|V|) \sim \mathcal{O}(i|E|)$.

PHAST [123] runs multiple SSSP on GPU concurrently. However, this work only applies to road network graphs and suffers from small-world graphs which do not have good separators as mentioned in [124]. Two MSSP algorithms [125, 126] mainly focus on special graphs, i.e., planar and embedded graphs, respectively. In contrast, our iBFS applies to both small-world and random graphs, and leverages the opportunity of frontier sharing to accelerate concurrent BFS.

Power efficiency [127] is of great importance to system design, e.g., [128] shuts down GPU streaming processors predictively to save power. Our work has shown that GPU-based graph algorithms have huge potential in delivering high performance and energy-efficiency.



(a) Per-level average



(b) Top-down, direction-switching and bottom-up

Figure 2.5: Boxplot of percentage of frontiers (a) for per-level average and (b) for top-down, direction-switching, and bottom-up.

2.4 Design Challenges

Challenge #1: Putting GPU Threads to Good Use

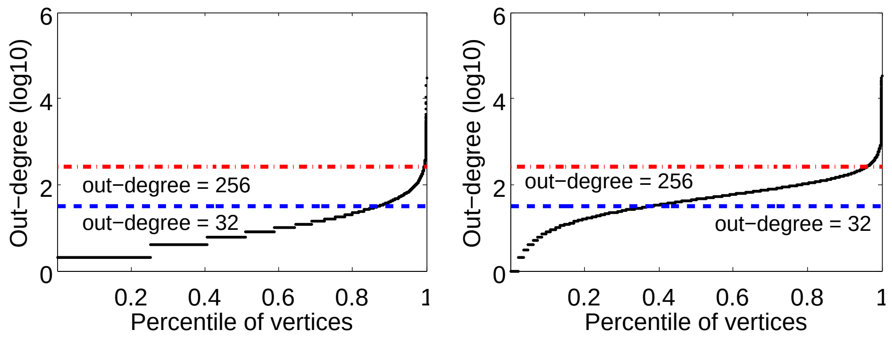
Eliminating the need of atomic operations on GPUs for the frontier queue generation which has been the focus of prior work [92, 74, 93, 91, 129, 106] solves only half of the problem - the queue shall consist of only the frontiers, that is, the vertices that need to be explored in next level. Using the status array for next level traversal, although avoiding atomic operations, would assign one GPU thread for each vertex, regardless whether it is a frontier [91, 92]. This inefficient approach would over-commit GPU threads because at most levels the majority of the vertices would not be a frontier. Alternatively, another work [74] generates the frontier queue with warp and historical culling, but again this approach could not completely avoid duplicated vertices across warps being enqueued. Figure 2.5(a) shows the boxplot of the percentage of the frontiers at each level across different graphs, where the mean and maximum percentage, as well as standard deviation, are presented. Note that the numbers here include the frontiers for both top-down and bottom-up directions. It turns out that the graphs have on average 9% frontiers per level with standard deviation of 15%. In particular, the R-MAT graph has the largest average ratio of 11% and maximum of 57%, while Twitter has the smallest average of 1% and maximum of 10.2%. If a thread were assigned to each vertex at every level, on average at least 31% of the threads would idle. Therefore, it is critical to have a queue that consists of the frontiers only, instead of wasting valuable GPU threads on those with no work to perform.

This challenge is further exacerbated by the need of direction switching between top-down and bottom-up, which generates the frontiers by focusing on two distinct sets of vertices (visited in top-down vs. unvisited in bottom-up). To illustrate this problem, we present the percentage of the frontiers by BFS traversal directions in Figure 2.5(b). In general, bottom-up levels have more frontiers than top-down, i.e., 1.5% vs 0.4%. In particular, the queue for the level when switching from top-down to bottom-up has most frontiers at 52% on average. Using the status array alone at this level would remain inefficient. The above observation leads us to develop Enterprise with new GPU threads scheduling that aims to prepare a frontier queue that is direction optimized for GPU memory hierarchy.

Challenge #2: Balancing Workloads Among GPU Threads

This challenge stems from that fact that large variance exists in the frontiers' out-degrees. If a frontier has more edges, the GPU thread assigned to it would naturally need to carry out more expansion and inspection. To illustrate this imbalance, we plot the CDF of the edge counts for two social networks in Figure 2.6, where the average out-degrees for Gowalla and Orkut are 19 and 72 respectively. In Gowalla, 86.7% and 99.5% of the vertices have fewer than 32 and 256 edges. In contrast, while Orkut has a smaller portion (37.5%) of the vertices with fewer than 32 edges, it has more (58.2%) with out-degree between 32 and 256. Furthermore, a fraction (0.5% and 4.2%) of vertices have more than 256 edges in Gowalla and Orkut with a long tail to around 30K edges.

Statically assigning one fixed number of threads (e.g., a warp or CTA) is inefficient because the per-level runtime would be dominated by the threads with heavy workload. Another inefficiency may also arise from the mismatch from the thread

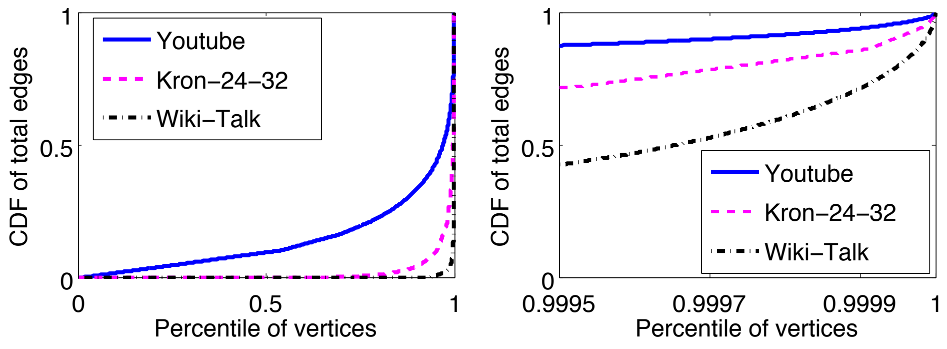


(a) Gowalla out-degree (b) Orkut out-degree

Figure 2.6: Cumulative Distribution Function (CDF) of out-degrees of vertices sorted by out-degree: (a) Gowalla (b) Orkut.

count and the workload. For example, if one CTA were assigned to work on a frontier with fewer than 32 adjacent vertices, more than 200 threads in this CTA would have no work to do. On the other extreme, some frontiers with very high out-degrees will require more than one CTA, e.g., some graphs we examine have vertices with up to 10^6 edges. To address this challenge, Enterprise introduces a new approach of classifying frontiers based on the out-degrees and assigning an appropriate GPU parallel granularity at runtime.

Challenge #3: Making Bottom-Up BFS GPU-Aware



(a) All range (b) Zoom into [0.9995, 1]

Figure 2.7: CDF of total edges in Youtube, Wiki-Talk and Kron-24-32 graphs. The vertices are sorted by out-degrees: (a) Vertices of all range (b) Zoom in range [0.9995, 1].

Implementing direction-optimizing BFS on GPUs is challenging by itself. Direction-optimizing BFS has first been proposed and implemented on multi-core CPUs in [70],

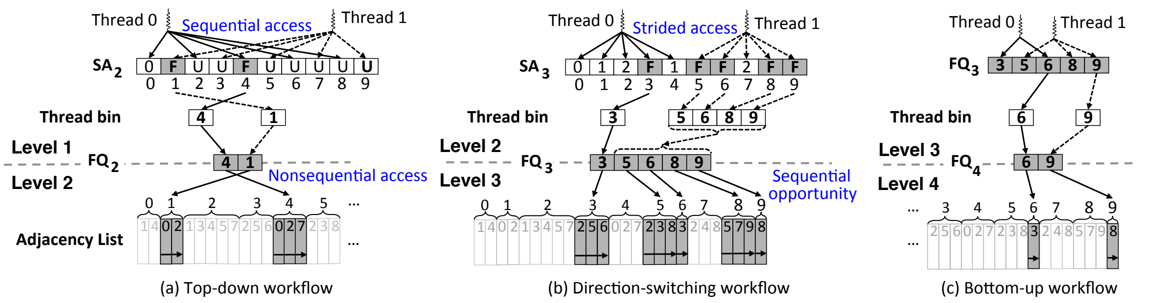


Figure 2.8: Streamlined GPU threads scheduling using the graph example from Figure 2.1, with three workflows: (a) top-down, (b) direction-switching at the explosion level, and (c) bottom-up. Sequential access means two threads access consecutive adjacent elements at each iteration. Strided access means two threads access elements in stride manner at each iteration.

but without further optimizations would not run efficiently on GPUs that can run thousands of threads but with a relatively smaller (e.g., 12GB) and slower (e.g., 200-400 cycles) global memory. In contrast, as previously shown in Table 1.2, modern CPUs have tens of cores and threads with large L3 cache and main memory with short access latency. Fortunately, what GPU lacks on global memory is compensated by a massive number of registers and software-configurable shared memory (L1 cache), which can be utilized to accelerate memory intensive algorithms like BFS.

The CPU-based bottom-up BFS uses the status array to supply the unvisited vertices for inspection, and direction switching between top-down and bottom-up depends on the numbers of unexplored edges. In Enterprise, the GPU-based bottom-up BFS leverages a small set of highly connected vertices called hub vertices. Formally, we define a hub vertex as follows.

Definition (Hub Vertex) Let v be a vertex of the graph G. Consider v be a hub vertex if its out-degree is greater than a threshold τ.

Here τ is graph specific, e.g., in the order of 100Ks for Twitter. It is common that a few hub vertices in power-law graphs connect to a great number of vertices. Figure 2.7 presents both the CDF of total edges and a zoom-in view for the range of [99.95%, 100%] of the vertices. For the YouTube graph, one can see that 330 hub vertices (i.e., 0.03% of the total vertices) contribute to 10% of the total edges. Similarly, 770 hub vertices (0.005%) in Kron-24-32 produce 10% of the total edges, and 96 hub vertices (0.004%) in Wiki-Talk account for 20% of the total edges.

The most unique features of our GPU-based bottom-up BFS are: 1) Enterprise switches the direction at what we refer in this paper as the *explosion level* where a large quantity of hub vertices need to be visited. In this work, we have found that the number of hub vertices in the frontier queue can serve as a better indicator for direction switching, which can easily implemented on GPUs. And more importantly, 2) caching hub vertices turns out to be very beneficial for bottom-up BFS.

2.5 Enterprise: GPU-Based BFS

2.5.1 Streamlined GPU Threads Scheduling

Combining the power of the status array and frontier queue, Enterprise is able to produce streamlined scheduling of GPU threads through generating the frontier queue at each level with a scan of the status array. At each level, Enterprise starts with identifying the frontiers and updating the status array in a manner similar to [91, 92]. Once this step completes, Enterprise dispatches GPU threads to scan the vertices in the status array. When a frontier is found, the thread will store this vertex in its own thread bin. All the thread bins are stored in the global memory. Next, prefix sum is used to calculate the offset of each bin in the frontier queue [130, 131]. Lastly, the frontiers in each bin are copied to the queue in parallel. In all, the benefits are clear in avoiding thread synchronization (from using the array) and reducing idle threads at the next level (from using the queue). However, as we have shown, BFS direction can lead to a large disparity in the number of frontiers at each level.

To this end, Enterprise schedules GPU threads using three queue generation workflows for top-down, direction-switching, and bottom-up, to optimize the memory accesses in all cases. The status array, frontier queue and adjacency list reside in GPU global memory, and accessing the global memory randomly would only achieve a meager 3% of sequential read bandwidth. To maximize the overall performance, it is critical that we optimize the access patterns at different stages of BFS.

Top-down workflow. In this direction, Enterprise uses the GPU threads to

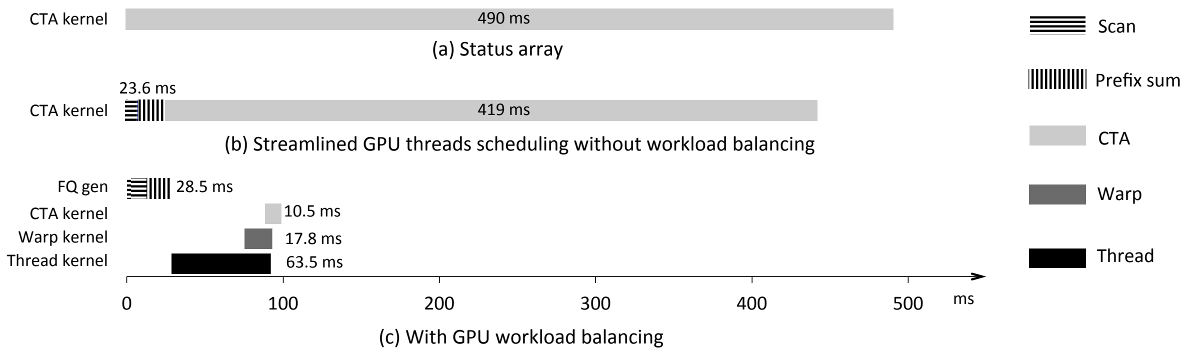


Figure 2.9: Execution timeline before and after streamlined GPU threads scheduling and workload balancing for the explosion level of Facebook.

scan the status array in an interleaved manner. For the example in Figure 2.8(a), two threads are dispatched at level 1: thread 0 checks the status of five vertices {0, 2, 4, 6, 8}, while thread 1 checks the others {1, 3, 5, 7, 9}. This division of work performs a sequential memory access of the status array. When prefix sum is completed, threads 0 and 1 will copy their own thread bins into FQ_2 concurrently. In this case, FQ_2 stores two frontiers out-of-order as {4, 1}, which will introduce nonsequential memory access at level 2, that is, BFS accesses the adjacency list of vertex 4 before vertex 1. Fortunately, the benefit of sequential access of the status array outweighs the potential drawback of random access of the adjacency list. For top-down, adjacent vertices in the status array are unlikely to become frontiers at the same level, as there are only a small number (average 0.4%) of frontiers, as shown in Figure 2.5(b).

Direction-switching (explosion-level) workflow. The situation is different in this case. Here the GPU threads are allocated a certain portion of the status array to scan. Using the same example, at level 2, again two threads will be used: this time thread 0 checks the status of five vertices {0, 1, 2, 3, 4} while thread 1 checks five vertices {5, 6, 7, 8, 9}. Unlike the top-down workflow, this approach would incur strided memory access during the scan. Next, prefix sum is performed on thread bins and in this example FQ_3 consists of {3, 5, 6, 8, 9}. The performance benefit comes from that the (bottom-up) frontiers may appear in order in the queue, which in turn leads to sequential memory access at the next level. At the explosion level, chances are that adjacent vertices are all unvisited, because most are as we have shown in Figure 2.5(b). This workflow takes advantage of this fact to accelerate the next level traversal, e.g., at level 4, loading the adjacent list of vertices 5 and 6 are sequential adjacent memory access, and so are vertices 8 and 9.

At the explosion level, this approach will spend average 2.4× more time to scan the status array, as compared to the top-down workflow. For example, using top-down workflow would take 0.57 ms for the explosion level on Hollywood. In contrast, using direction-switching workflow will take a longer time of 0.86 ms. But this approach will improve the performance of next level traversal by average 37.6%, e.g., Hollywood runtime at the level right after the explosion decreases from 2.7 to 2 ms. When combined, because the latter step takes longer wall clock time, the overall performance achieves an average speedup of over 16% across all the graphs, with the best improvement of 33% on Facebook.

Bottom-up workflow. The key insight is that for bottom-up, the queue for the current level is always a subset of the previous queue, as the frontiers are always unvisited vertices. Instead of continuing to use the status array, we directly use the frontier queue from the preceding level to generate the queue for current level shown in Figure 2.8(c). This is done by simply removing the vertices that belong to current level. This approach eliminates the need of scanning the whole status array. Only a small (and fast shrinking) subset is inspected at each level. For example, at level 3, FQ_4 is created by removing vertices $\{3, 5, 8\}$ from FQ_3. Our tests show this approach delivers 3% improvement across various graphs.

To summarize, this technique increases the number of GPU threads that actively work on frontiers and issue memory load/store requests, which we will see in the experiments that the utilization of memory load/store function unit increases dramatically. Using this design, the queue can still be generated very quickly from 2.2 to 53.7 ms for different graphs, which accounts for about 11% of the overall BFS traversal time, yet delivers 2× to 37.5× speedup. Figure 2.9 presents an execution trace of BFS execution for the explosion level of Facebook. Clearly, despite generating the frontier queue takes 23.6 ms, because a good workqueue is prepared, new threads

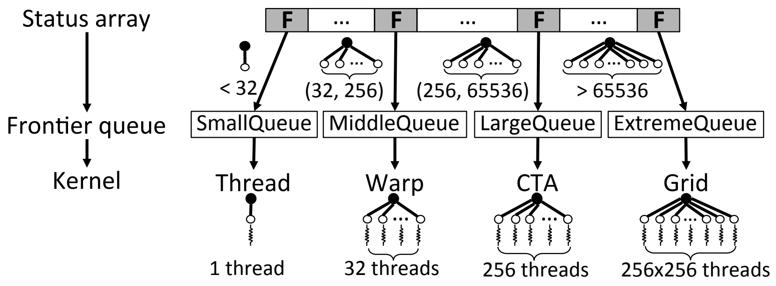


Figure 2.10: GPU workload balancing.

scheduling reduces the runtime of expansion and inspection from 490 ms to 419 ms, a net saving of 46 ms.

2.5.2 GPU Workload Balancing

Now that Enterprise can generate a good frontier queue quickly, but the benefit would be minimal if the queue would lead to imbalanced workload. In this work, we believe that parallel granularity of GPU shall be leveraged when scheduling work from the frontier queue to ensure high thread-level parallelism. Ideally, each thread, regardless of standalone, within a warp or CTA, shall have an equal amount of work (expansion and inspection) at each level. To achieve this goal, Enterprise classifies the frontiers based on their out-degrees (potential workload) and allocates a matching parallel granularity. Enterprise focuses on the use of threads over warps or CTAs, different from prior work [73, 74]. This is motivated by the fact that the majority of the vertices in a graph have small out-degrees. For the graphs studied, the average percentage of the vertices with fewer than 32 edges is 68% and may go as high as 96% in Twitter.

Enterprise classifies the frontiers that are generated with the previous technique into four queues, SmallQueue, MiddleQueue, LargeQueue and ExtremeQueue, based on the out-degrees of each frontier. Specifically, the frontiers in SmallQueue have fewer than 32 edges, MiddleQueue between 32 and 256, LargeQueue between 256 and 65,536 and ExtremeQueue more than 65,536. During frontier queue generation, each thread puts the discovered frontiers into one of four thread bins according to their out-degrees. At the next level, four kernels (Thread, Warp, CTA and Grid) with different number of threads will be assigned to work on different frontier queues in order to balance the workloads among the threads, as shown in Figure 2.10. All kernels are executed concurrently with Hyper-Q support.

We maintain an ExtremeQueue for dealing with vertices with extremely high out-degrees. For instance, one vertex in KR2 has over 2.5 million edges. If one CTA were assigned to inspect this vertex, it would require more than 10,000 iterations. This type of vertex exists for many graphs as we have seen as long tails in Figure 2.6 and 2.7. Subsequently, expanding from these vertices would require rather long runtime, which without special handling may greatly prolong the traversal of the whole level. Using the whole grid here can considerably speed up the execution, e.g., 1.6$\times$ speedup is achieved on KR0.

In Figure 2.9(b) and (c), one can see the changes in runtime before and after workload balancing. Again, although this optimization adds another 5 ms of overhead to classify the frontiers, we are able to shorten the overall runtime drastically, from 419 ms to 76.5 ms. In particular, the Thread kernel takes 63.5 ms, Warp 17.8 ms, and CTA kernel 10.5 ms, where there is significant overlapping among the three kernels. In short, this technique further removes idling threads in each CTA and warp compared to prior methods, which similar to the first technique will lead to higher utilization on GPU memory units.

2.5.3 Hub Vertex Based Optimization

Direction-switching parameter. In this work, we have found that it is cumbersome to tune the parameter α to determine when to switch from top-down to bottom-up. Instead, as hub vertices make up a good portion at the explosion level, we propose to use the ratio of hub vertices in the frontier queue as an indicator for direction switching. We define the parameter γ formally as:

$$\gamma = \frac{F_h}{T_h} \times 100\% \tag{2.1}$$

where F_h is the number of hub vertices in the frontier queue (collected per level) and T_h represents the total number of hub vertices, which can be calculated very quickly

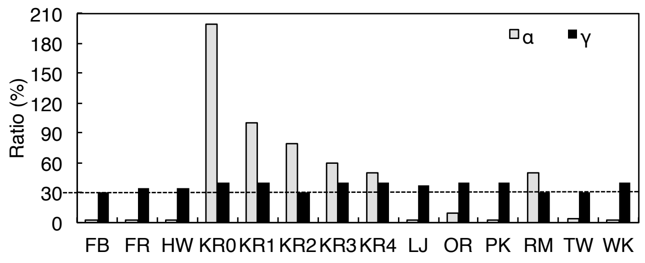


Figure 2.11: Comparison of direction-switching parameters.

at the first level. Our experiment shows that γ is stable without the need for manual tuning. Figure 2.11 shows that all graphs should switch direction when $\gamma \in (30, 40)\%$, a very small range compared to α that fluctuates between 2 and 200. In this work, we set the direction-switching condition as γ being larger than 30.

Enterprise traverses on average 4 levels top-down and 8 levels bottom-up across various graphs, about one level sooner than prior method [70]. For the Kronecker graphs, using α would inspect 4% and 17% of the edges in top-down and bottom-up respectively, avoiding to visit the remaining 79% edges. Using our hub vertex based parameter γ would inspect 1% and 36% edges in top-down and bottom-up. At first glance, Enterprise would inspect more edges in total, hurting the performance. Fortunately, as we have shown, direction switching happens at the explosion level that is dominated by hub vertices, and bottom-up traversal focuses on identifying the edges connecting the frontiers to recently visited hub vertices. As a result, a good cache of hub vertices lends itself nicely for both scenarios.

Hub vertex cache. In this work, we propose to cache hub vertices in GPU shared memory during direction switching and bottom-up, which can greatly reduce the overhead of random global memory accesses. This benefit is achieved because large amount of frontiers in bottom-up are very likely to connect to hub vertices. However, as GPU shared memory is small (64 KB), we need to carefully balance the number of hub vertices cached and the occupancy of the GPU that is defined as the ratio of active warps running on one SMX and the maximum number of warps that one SMX can support theoretically (64). If a grid contains 256 $\times$ 256 threads, the full occupancy of K40 means 8 CTAs running on one streaming processor and thus each CTA only has 6 KB shared memory to construct a cache holding around 1,000

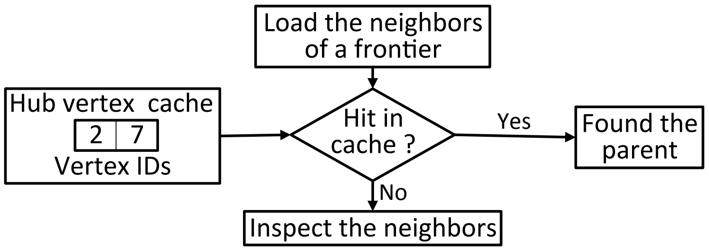


Figure 2.12: Hub vertex cache design, using the level 4 traversal in example graph from Figure 2.1.

hub vertices.

Hub vertex cache (HC) is implemented in two steps. First, during the frontier queue generation, Enterprise caches the vertice IDs of those have just been visited at the preceding level and also with high out-degrees. We use a hash function to figure out which index to store each vertex ID, that is, $HC[hash(ID)] = ID$. Second, during the frontier identification, Enterprise will load the frontier's neighbors, and check whether the vertex ID of any neighbor is cached. If so, the inspection will terminate early with the cached neighbor identified as the parent for this frontier. In this case, the cache avoids accessing this neighbor's status in the global memory.

Figure 2.12 presents the workflow of the hub vertex cache. In this example, Enterprise puts vertice IDs {2,7} in the hub vertex cache because these two vertices are visited in the preceding level and with the high out-degrees. At the current level, Enterprise will load the neighbors {2, 5, 6} of vertex 3 for inspection. As vertex 2 is cached, Enterprise will mark vertex 2 as the parent of vertex 3 and terminate the inspection. On the other side, if a frontier like vertex 6 does not have a cached neighbor, Enterprise will continue to inspect the statuses of its neighbors that reside in the global memory. As shown in Figure 2.13, the hub vertex cache is very effective on various graphs, saving 10% to 95% of global memory accesses. It is worthy to point out that caching hub vertices has limited benefit for top-down BFS, as it likely encounters very few hub vertices.

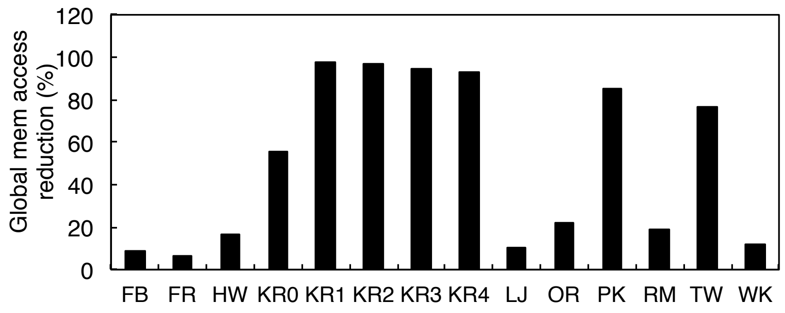


Figure 2.13: Global memory accesses reduced by hub cache.

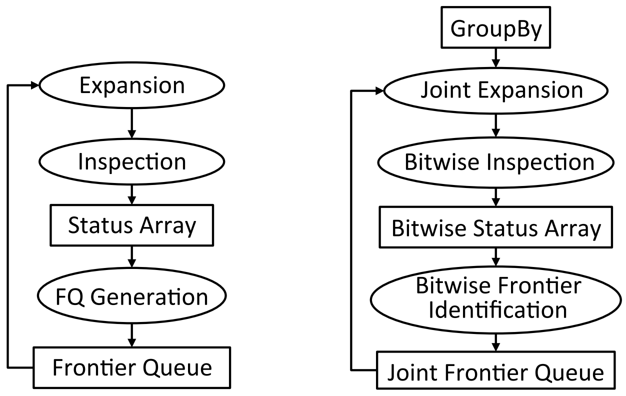
2.5.4 Multi-GPU Enterprise

Enterprise exploits 1-D matrix partition method [132] to distribute the graphs across multiple GPUs. Specifically, each GPU is responsible for an equal number of vertices from the graph, and thus a similar number of edges. We leave the study of 2-D partition as future work. During traversal, Enterprise proceeds in three steps: (1) Each GPU identifies the current level vertices in a private status array by expanding from a private frontier queue. (2) All the GPUs communicate their private status arrays to get the global view of most recently visited vertices. In this step, each GPU uses a CUDA instruction `__ballot()` to compress the private status array into a bitwise array where a single bit is used to indicate whether one vertex is just visited. This compression reduces the size of communication data by 90%. (3) Each GPU scans the updated private status array to generate its own private frontier queue.

2.6 iBFS: GPU based Multi-Source BFS

In a nutshell, iBFS as shown in Figure 2.14, consists of three unique techniques, namely, joint traversal, GroupBy, and bitwise optimization, which will be discussed in Section 2.6.1, 2.6.2, and 2.6.3, respectively.

Ideally the best performance for iBFS would be achieved by running all i BFS instances together without GroupBy. Unfortunately, ever-growing graph sizes, combined with limited GPU hardware resources, puts a cap on the number of concurrent BFS instances. In particular, we have found that GPU global memory is the dominant



(a) A Single BFS (b) iBFS

Figure 2.14: The flow charts of (a) BFS, (b) iBFS.

factor, e.g., 12GB on K40 GPUs compared to many TB-scale graphs.

Let M be GPU memory size and N the maximum number of concurrent BFS instances in one group (i.e., the group size). If the whole graph requires S storage, a single BFS instance needs $|SA|$ to store its data structures (e.g., the status array for all the vertices), and for a joint traversal each group requires at least $|JFQ|$ for joint data structures (e.g., the joint frontier queue), then $N \leqslant \frac{M-S-|JFQ|}{|SA|}$. In most cases N satisfies $1 < N \ll i \leqslant |V|$. In this paper, we use a value of 128 for N by default.

Unfortunately, randomly grouping N different BFS instances is unlikely to produce the optimal performance. Care has to be taken to ensure a good grouping strategy. To illustrate this problem, for a group A with two BFS instances BFS-s and BFS-t, let $JFQ_A(k)$ be the joint frontier queue of group A at level k, $FQ_s(k)$ the individual frontier queue for BFS-s, and FQ_t for BFS-t. Thus, $|JFQ_A(k)| = |FQ_s(k)| \cup |FQ_t(k)| - |FQ_s(k)| \cap |FQ_t(k)|$, where $|FQ_s(k)| \cap |FQ_t(k)|$ represents the shared frontiers between two BFS instances. Clearly, the more shared frontiers each group has, the higher performance iBFS will be able to achieve. Before we describe the GroupBy technique in Section 2.6.2 that aims to maximize such sharing within each group, we will first introduce how iBFS achieves joint traversal in the next section which makes parallel execution possible.

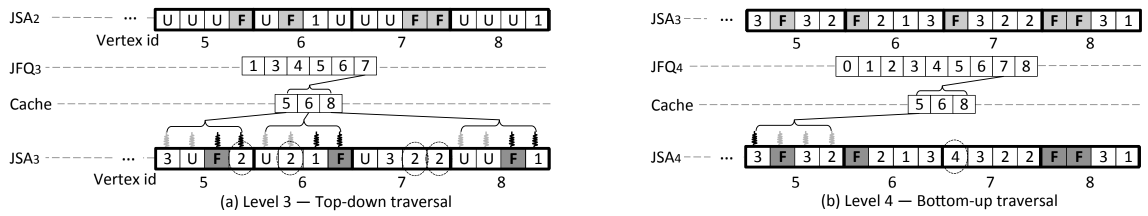

JSA2
Vertex id
JFQ3
Cache
JSA3
(a) Level 3 — Top-down traversal
JSA3
JFQ4
Cache
JSA4
(b) Level 4 — Bottom-up traversal

Figure 2.15: iBFS traversal on joint status array (a) Level 3 – top-down traversal and (b) Level 4 – bottom-up traversal using the example graph and four BFS instances from Figure 2.3, here i=4 and black threads are active while gray are not. While this figure only presents the inspection of one vertex 7, we keep JSA updated with the latest depth represented as dotted circles. Note that JSA_3 is different between (a) top-down and (b) bottom-up as frontiers are identified differently.

2.6.1 Joint Traversal

In this work, we propose to implement iBFS in a single GPU kernel, different from prior GPU-based work [79]. This way, iBFS will be able to exploit the sharing across different BFS instances, e.g., if two threads of the same kernel are scheduled to work on a shared frontier, iBFS only needs to load adjacent vertices from global memory once. This benefit would be, otherwise, not possible on GPUs with a multi-kernel implementation which would be used for the aforementioned naive implementation.

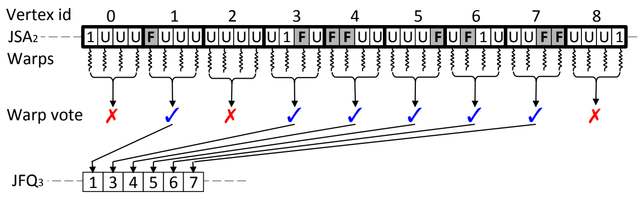


Figure 2.16: Generate joint frontier queue (JFQ_3) from joint status array (JSA_2). Assuming we are executing the four BFS traversals of Figure 2.3 in a single kernel.

The joint traversal of iBFS uses two joint data structures for all concurrent BFS instances: (1) **Joint Status Array (JSA)** is used to keep the status of each vertex for all instances. For each vertex, iBFS puts its statuses for different BFS instances sequentially. For example, in Figure 2.16, as we run four concurrent BFSes, four bytes are used for each vertex. For vertex 0, the first byte of 1 indicates that vertex 0 has been visited with the depth of 1 in BFS-0, and the next three bytes of U indicate that the same vertex 0 has yet been visited for all three other BFS instances. (2) **Joint Frontier Queue (JFQ)** is a set of all the frontiers from concurrent BFS instances, where any shared frontier appears only once. Thus, this queue requires the maximum size of $|V|$ to store the frontiers for all i BFS instances. In comparison, using a private frontier queue would require a much bigger queue with the size of $i \times |V|$.

A GPU uses single instruction multiple thread (SIMT) model and schedules threads in a *warp* that consists of 32 threads [9]. Several warps can form a thread block called *Cooperative Thread Array* (CTA). The threads within a block can communicate with each other quickly with *shared memory* and built-in primitives.

To generate the JFQ, iBFS assigns one warp to scan the status of each vertex as in Figure 2.16. If this vertex happens to be a frontier for any BFS instance, iBFS needs to store it in the joint frontier queue, e.g., vertex 1 is put in JFQ_3 because it is a frontier for BFS-0 at level 3. In contrast, vertex 0 is not considered a frontier for all four BFS instances.

It is worthy to note that iBFS uses a CUDA vote instruction, i.e., `__any()`, to communicate among different threads in the same warp and schedules one thread to enqueue the frontier. Furthermore, iBFS uses another CUDA feature `__ballot(parameter)` to generate a separate variable to indicate which BFS instances share this frontier. This is important for shared frontiers, e.g., while vertex 7 is a frontier in BFS-2 and BFS-3, it would appear once in the joint frontier queue. Removing such redundant frontiers from the queue substantially reduces the number of costly global memory updates, which contributes partly to performance gains obtained by iBFS.

iBFS's joint traversal introduces two unique memory optimizations to reduce the number of expensive global memory transactions: (1) during expansion, iBFS uses a new cache presented in Figure 2.15 to load the adjacent vertices of a frontier from GPU's global memory to its shared memory to feed all BFS instances. This way the neighbors from each frontier will only be loaded from global memory once, although numerous requests may come from multiple BFS instances. This benefit is not limited just to shared frontiers, rather every frontier in the queue. And (2) during inspection, iBFS schedules multiple threads with contiguous thread IDs to work on each frontier. Here the number of threads is the same as the that of concurrently executed BFS instances. This is because on GPUs one global memory transaction typically fetches

16 contiguous data entries from an array and only continuous threads can share the retrieved data. On the other hand, if continuous threads write to the same memory block, such writes are coalesced into one global memory transaction as well.

Now we will use an example to show how iBFS utilizes these new structures in both top-down and bottom-up traversal. Figure 2.15(a) exhibits the top-down traversal of vertex 7 which is a frontier in the proceeding level (i.e., level 2). During expansion, the neighbors {5, 6, 8} of vertex 7 are loaded in the cache. During inspection, BFS instances that do not have this frontier will not inspect the neighbors, specifically, the first and second threads in this figure. On the other hand, when the third and fourth threads access the statuses of three neighbors, it is performed in a single global memory transaction since these statuses are stored side by side and accessed by contiguous threads. At this level, vertex 7 will also have its status updated with the depth of 2.

The bottom-up traversal is performed in a different manner as shown in Figure 2.15(b). For the frontier vertex 7, iBFS will similarly load its adjacent vertices {5, 6, 8} into the cache. The difference here is that iBFS will check if {5, 6, 8} are already visited, if so, mark the depth of 7 as 4.

Note that a vertex can be a frontier at both top-down and bottom-up levels, for example, vertex 7 in Figure 2.15. In Figure 2.15(a), vertex 7 is a frontier for top-down traversal of the third and fourth BFSes, and in Figure 2.15(b) bottom-up of the first BFS. Clearly, as long as one BFS considers a vertex as a frontier at a particular level, this vertex shall be enqueued in iBFS.

2.6.2 GroupBy

In this section, we will introduce the concept of sharing ratio, and propose several outdegree-based GroupBy rules. We will first use top-down traversal for the discussion of GroupBy and later present the impacts on bottom-up BFS.

Frontier Sharing Degree and Ratio

To a great extent, the performance of iBFS is determined by how many frontiers are shared at each level during the joint traversal of each group. This is because if multiple BFS instances have a common frontier in one level, all its edges will be checked only once, thus leading to overall performance improvement. In this work we define for any group, say group A with the size N, the Sharing Degree (SD$_A$) as the degree of sharing that exists in the joint frontier queue:

$$SD_A = \frac{\sum\limits_k \sum\limits_{j=1}^{N} |FQ_j(k)|}{\sum\limits_k |JFQ_A(k)|} \tag{2.2}$$

where $j \in$ [1, N] and represents the j-th BFS instance in the group A, and k represents the level (or depth) for the traversal. In essence, SD$_A$ shows on average each joint frontier is shared by how many BFS instances in a group. Thus, the Sharing Ratio can be easily calculated as SD divided by the total number of instances in the group.

Traditionally, the time complexity of a BFS is calculated by the number of inspections performed, that is, the edge count $|E|$. Thus the time of a sequential execution of group A will be N $\cdot$ $|E|$.

In this work, We use $T_A(k)$ to represent the time of the joint execution of group A at the level k, that is,

$$T_A(k) = \sum_{v \in JFQ_A(k)} outdegree(v)$$

where v stands for each frontier in the JFQ at the level k. And the total runtime T_A for group A can be calculated by summarizing the runtime of each level, which is:

$$T_A = \sum_k T_A(k) = \sum_k \sum_{v \in JFQ_A(k)} outdegree(v) \tag{2.3}$$

According to the Amdahl's Law, the speedup $Speedup_A$ of joint traversal for group

A over sequential execution is:

$$Speedup_A = \frac{N \cdot |E|}{T_A} \tag{2.4}$$

Let $\bar{d}$ be the average outdegree, we can obtain the expected value of $Speedup_A$ using equations (2.3) and (2.4). Thus,

$$\begin{aligned} \mathbb{E}[Speedup_A] &= \mathbb{E}[\frac{N \cdot \bar{d} \cdot |V|}{\bar{d} \cdot \sum_k |JFQ_A(k)|}] \\ &= \mathbb{E}[\frac{N \cdot |V|}{\sum_k |JFQ_A(k)|}] \end{aligned} \tag{2.5}$$

Lemma 1. *For any BFS group, say group A, its sharing ratio is equal to the expected value of $Speedup_A$, that is,*

$$SD_A = \mathbb{E}[Speedup_A] \tag{2.6}$$

Proof. Because for the j-th BFS every vertex will become a frontier at one of the levels, the sum of $|FQ_j(k)|$ across all levels is equal to the total number of the vertices, that is, $\sum_k |FQ_j(k)| = |V|$.

$$\begin{aligned} SD_A &= \frac{\sum_k \sum_{j=1}^{N} |FQ_j(k)|}{\sum_k |JFQ_A(k)|} = \frac{\sum_{j=1}^{N} |V|}{\sum_k |JFQ_A(k)|} \\ &= \mathbb{E}[Speedup_A] \end{aligned} \tag{2.7}$$

□

Lemma 1 demonstrates that the sharing ratio reflects the performance of the joint execution when it is compared to the sequential execution of such a group. However, for any group, since the size of JFQ at each level is not known until runtime, we will not able to calculate a priori the sharing ratio as well as the expected performance. Fortunately, we discover that the sharing ratios at the first few levels can be used as a good metric, and there also exists an important correlation for sharing ratios at consecutive levels.

Theorem 1. *Given any two BFS groups A and B with the same number of BFS instances, if at the level k their sharing ratios obeys $SD_A(k) > SD_B(k)$, at the level $k+1$ the following relationship, $\mathbb{E}[SD_A(k+1)] > \mathbb{E}[SD_B(k+1)]$, will hold.*

Proof. Let us start with one group A. At the level k, the j-th BFS performs two main tasks for a frontier vertex v, i.e., expansion and inspection. In the first task, the j-th BFS fetches the neighbor list of frontier v, and in the second task, it checks all neighbors. The number of inspections is equal to the outdegree of v.

We define *null inspections* as those that do not lead to an increase in the size of the frontier queue at a particular level. Note that as vertices are shared, null inspections do not necessarily mean such inspections have not found a frontier at the next level. There are three cases that render a neighbor check as a null inspection. The first two cases are relevant to a single BFS, while all three are applicable to iBFS. For a neighbor w of the frontier v,

Case 1 : the neighbor w is visited.

Case 2 : the neighbor w has already been marked as a new frontier by other inspections at the level k. This check will be discarded, and will not increase the size of the frontier queue. This happens because w may have additional parents other than v.

Case 3 : the neighbor w has already been marked as a new frontier by other concurrent BFS instances. Similarly, in joint traversal, this check will be discarded, and will not increase the size of the frontier queue, because w may be shared by concurrent BFS instances.

We use three values to represent the occurring percentages of three cases: α accounts for case 1 and 2 for a single BFS, β for case 1 and 2 for iBFS, and γ for case 3 for iBFS alone. Thus, the probability of the complement of case 1 and 2 for a single BFS can be represented by $(1-\alpha)$, and so on.

Now for the level $k+1$, the size of JFQ is equal to the sum of the number of frontiers shared by different BFS instances within group A, from one to N instances.

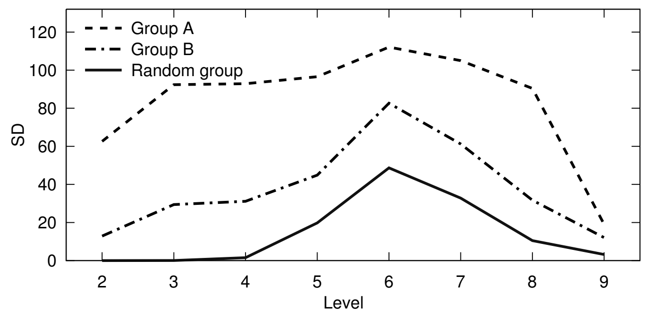


Figure 2.17: Sharing ratio trend of Facebook graph (FB).

Thus, let $s_j(k)$ be the number of vertices that are shared by exactly j BFS instances, we have

$$SD_A(k+1) = \frac{\sum_{j=1}^{N} |FQ_j(k+1)|}{|JFQ_A(k+1)|} = \frac{\sum_{j=1}^{N} j \cdot s_j(k+1)}{\sum_{j=1}^{N} s_j(k+1)} = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot d_j \cdot (1-\alpha_j)}{\sum_{j=1}^{N} s_j(k) \cdot d_j \cdot (1-\beta_j-\gamma_j)} \tag{2.8}$$

where d_j denotes the average out-degree of these frontiers.

Since we do not know a priori the values for d_j, α_j, β_j, and γ_j, if replacing them with average values $\bar{d}$, $\bar{\alpha}(k)$, $\bar{\beta}(k)$, and $\bar{\gamma}(k)$, the expected value of $SD_A(k+1)$ is

$$\mathbb{E}[SD_A(k+1)] = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot \bar{d} \cdot (1-\bar{\alpha}(k))}{\sum_{j=1}^{N} s_j(k) \cdot \bar{d} \cdot (1-(\bar{\beta}(k))-\bar{\gamma}(k))} \tag{2.9}$$

We assume that each single BFS has the same probability $\bar{\alpha}(k)$, so for iBFS, we have $\bar{\beta}(k) = \bar{\alpha}^j(k)$, and

$$\mathbb{E}[SD_A(k+1)] = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot \bar{d} \cdot (1-\bar{\alpha}(k))}{\sum_{j=1}^{N} s_j(k) \cdot \bar{d} \cdot (1-(\bar{\alpha}^j(k))-\bar{\gamma}(k))} \approx SD_A(k) \cdot \frac{1-\bar{\alpha}(k)}{1-\bar{\gamma}(k)} \tag{2.10}$$

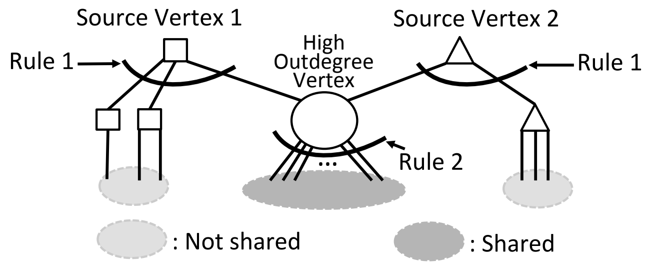


Figure 2.18: An illustration of a power-law graph.

Now we consider two groups A and B. If $SD_A(k) > SD_B(k)$, as $\bar{\alpha}(k)$ and $\bar{\gamma}(k)$ are independent of groups, we get $\mathbb{E}[SD_A(k+1)] > \mathbb{E}[SD_B(k+1)]$ from equation (2.10). $\square$

Figure 2.17 exhibits average sharing ratios for three different groups. We start from the second level as no BFS instances share the source vertices, and the maximum SD is equal to N, that is, 128. Since group A has a higher SD at the second level than group B, it always has higher ratios in the following levels. Similarly, group B has higher ratios than random. Clearly, as shown in Figure 2.17, sharing ratios would not increase monotonically for a group, which tend to peak at the first several bottom-up levels. Nevertheless, Theorem 1 implies that a higher sharing ratio of the initial levels can lead to a higher expected sharing ratio in later levels. Combining Lemma 1 and Theorem 1, we can see that a GroupBy strategy can achieve good speedup by focusing on the first several levels as described in Lemma 2.

Lemma 2. *For two BFS groups A and B, for a small number k, if their initial sharing ratios obey $SD_A(k) > SD_B(k)$ at the level k, the expected values for the speedups will follow $\mathbb{E}[speedup_A] > \mathbb{E}[speedup_B]$.*

Outdegree-based GroupBy Rules

Lemma 2 states that a good GroupBy rule can be obtained by focusing on increasing the BFS sharing in the first several levels. Fortunately, an easy analysis on the outdegrees of the frontiers at these levels, coupled with a quick determination on the connectivity to *high-outdegree vertices*, can lead to two simple yet powerful **outdegree-based rules**:

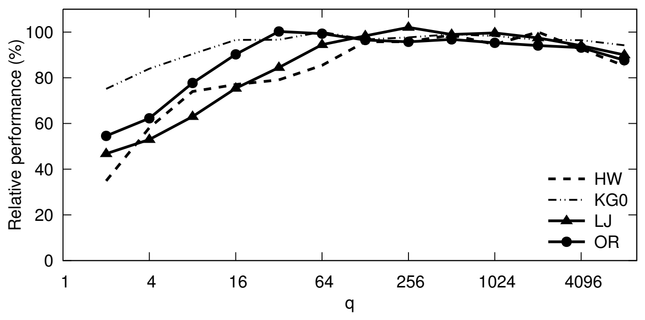


Figure 2.19: Performance of *GroupBy* for different q.

Rule 1 The out-degrees of these two source vertices are less than p.

Rule 2 Two source vertices connect to at least one common vertex whose outdegree is greater than q.

Both rules are complementary to each other. Since the first rule ensures small outdegrees of the source vertices, other non-shared neighbors will not amortize the sharing ratio contributed by the shared vertex with high outdegrees from the second rule. With these two GroupBy rules, two BFS instances will likely get high sharing ratio.

Figure 2.18 shows an example of a power-law graph which is the focus of this work. In this example, many vertices are connected to a high-outdegree vertex, and the source vertices for different BFSes are not exception. When they share a common high-outdegree vertex, high sharing ratio across multiple BFSes can be easily achieved. It is not required that the source vertex directly connects to a high-outdegree vertex, as long as within the first several levels. Simply put, for two BFSes, it is beneficial to group them together if their source vertices have relatively small number of edges and also connect to high-outdegree vertices.

Selection of p and q. Figure 2.19 plots the GroupBy performance for different q values. One can see that the performance rises initially and reaches the peak, typically around the range of 128 – 1024. The lower performance is observed for both smaller and larger q. For smaller q, the groups would likely have small sharing ratios. On the other hand, for larger q, very few BFS instances would satisfy the GroupBy rules. In this paper q is 128 by default. Once q is decided, p is selected in the ascending order

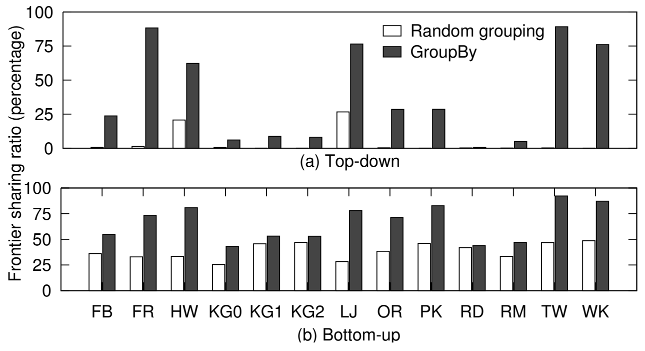


Figure 2.20: Frontier sharing ratio comparison between random and GroupBy.

from a sequence of numbers that are the power of 2. Here we select p from a sequence of $4, 16, 64$, and 128.

The rules are applied as follows. First, iBFS selects all groups that satisfy both rule 1 and 2 with pre-determined p and q. iBFS will run such groups directly when their sizes are larger than N, the maximum group size described in Section 2.6. Otherwise, several small groups, likely using different values of p, will be combined and run together. Second, iBFS will try to combine the groups with different high-outdegree vertices. Last, when no BFS satisfies both rules, iBFS will group the remaining them in a random manner.

Sharing ratio improvement. Figure 2.20 plots the improvement on the sharing ratio using outdegree-based rules for both top-down and bottom-up. Specifically, for top-down, our GroupBy rules improves the sharing ratio by $10\times$, which increases from average 3.9% to 39.3% for 128 BFS instances. For bottom-up, although the relative improvement is smaller, the sharing ratio is greatly improved to average 66.1% which we will discuss shortly.

For random graph that has a relatively uniform outdegree distribution, iBFS can adopt a slightly different rule. Since Theorem 1 and Lemma 2 still apply, iBFS can select a group of BFS instances if they share some common vertices from the sources. Our evaluation shows that such a rule may obtain $3.5\times$ and 5% improvement in top-down and bottom-up respectively on a random graph, albeit much smaller compared to other graphs in our test suite.

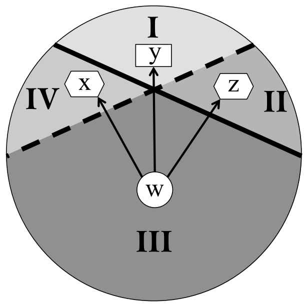


Figure 2.21: The circle represents all the vertices in a graph. There are two instances BFS-s and BFS-t. Area I and II represent the visited vertices for BFS-s (frontiers at top-down), and Area III and IV the unvisited vertices for BFS-s (frontiers at bottom-up). Similarly, Area I and IV represent the visited vertices for BFS-t, and Area II and III the unvisited vertices for BFS-t.

GroupBy on Bottom-Up BFS

So far we have focused on GroupBy during the top-down stage of BFS. Surprisingly, good GroupBy rules for top-down will lend themselves to achieving great speedups during the bottom-up stage. Together, GroupBy further accelerates iBFS performance by increasing frontier sharing and more importantly balancing the workload across various BFS instances. These two effects can be best explained using an example of two BFS instances, say BFS-s and BFS-t, within a group. In Figure 2.21, Area I represents shared, visited vertices between two BFSes whereas Area III shared, unvisited vertices. When BFS-s and BFS-t share more frontiers at top-down, that is, bigger Area I, they share more frontiers at bottom-up, that is, bigger Area III. From Figure 2.20(b), one can see that for bottom-up, all the graphs still achieve 66.1% on average, close to 2 times improvement. This is significant because in bottom-up frontier sharing ratio is already high (38.7%) to begin with.

The most significant benefit of GroupBy when it comes to bottom up lies on the fact that it helps balance the workload. For a shared vertex w in Area III shown in Figure 2.21, depending on which BFS it belongs to and who is its parent, it may need to search all three other areas. Again, When BFS-s and BFS-t share more frontiers at top-down, that is, bigger Area I, this also increases the likelihood that both BFS instances discover the shared parent y in Area I. As bottom-up inspection termi-

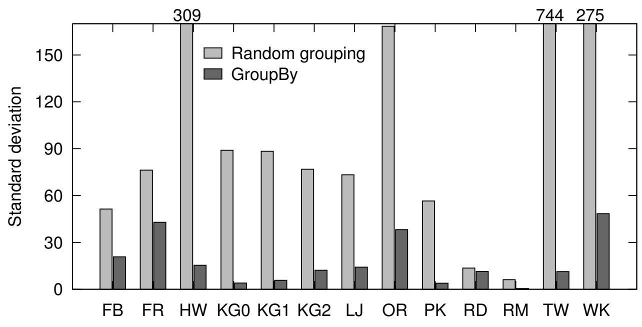


Figure 2.22: Standard deviation of the distribution for number of inspections during bottom-up before and after GroupBy.

nates as soon as a parent is found, such sharing will further lead to similar runtimes across different BFSes, that is, reducing the variance in runtimes. To demonstrate this, Figure 2.22 shows the standard deviation for the number of inspections during bottom-up before and after GroupBy. Since GroupBy combines the BFS instances that would find their parent with a similar runtime, it lowers the standard deviation by 13×. Among all the graphs, it helps the TW graph the most – by 66×, reducing the number of inspections from 744 to 11.3. Clearly, GroupBy helps to transform a highly imbalanced workload to a much more balanced one.

2.6.3 GPU-based Bitwise Operations

Although joint status array removes random inspections on shared frontiers, assigning a warp of threads to work on each frontier is not feasible when a graph has millions of frontiers. While modern GPUs like NVIDIA K40 provide thousands of hardware threads, clearly we still need to find a smart way to utilize threads effectively when dealing with several millions of vertices in a graph. To this end, we propose a novel concept of **Bitwise Status Array (BSA)** in iBFS that uses a single bit to represent the status of each vertex for different BFS instances. And all bits of one vertex are kept in a single variable. If this vertex is visited, we set it as 1, otherwise 0.

Figure 2.23 presents the mapping between joint and bitwise status array. Here for any vertex, while JSA uses the first four variables (3, U, 3, 2) to store the status for four BFS instances, BSA only needs a single variable (1011).

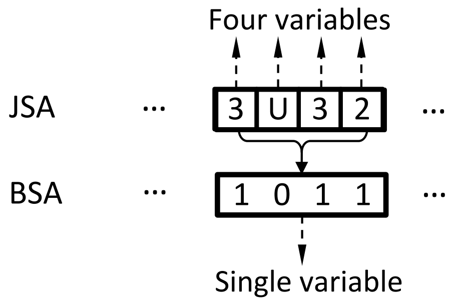


Figure 2.23: Mapping from joint status array (JSA) to bitwise status array (BSA) for one vertex.

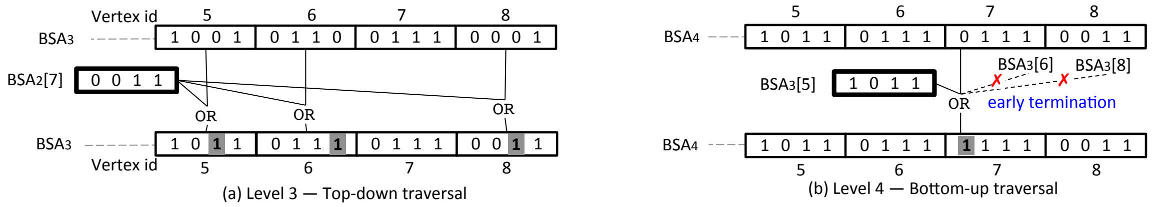

Vertex id
BSA3
BSA2[7]
OR
(a) Level 3 — Top-down traversal
BSA4
BSA3[5]
BSA3[6]
BSA3[8]
early termination
(b) Level 4 — Bottom-up traversal

Figure 2.24: Traversing four BFS instances of the example graph from Figure 2.3 with bitwise status array: (a) Level 3 – Top-down traversal. (b) Level 4 – Bottom-up traversal. Shadow bits stand for updated status.

With the bitwise status array, iBFS only needs one expansion thread to fetch the statuses of each vertex for all concurrent BFS. In addition, the inspection of concurrent BFS that will be described shortly can be executed by a single bitwise operation, which can be easily done with a single thread. In summary, this design frees up a substantial number of threads and further reduces the number of global memory access.

Bitwise Inspection. At level $k+1$, iBFS keeps a copy of bitwise status array – BSA_k. During traversal, the statuses for the most recently visited vertices are also marked as 1 in BSA_{k+1}. At the end of each level, the differences between BSA_k and BSA_{k+1} are used to identify the just visited vertices. Algorithm 1 shows the pseudo-code.

For top-down traversal, iBFS assigns a single thread to work on frontier f. This thread loads frontier's status (i.e., BSA$_k$[f]) from BSA$_k$ and subsequently sets all neighbors' status in BSA$_{k+1}$[v] via a bitwise `OR` operation. For each neighbor, this `OR` operation only affects the bits for the corresponding BFS instances that share this frontier, because only these bits in BSA$_k[f]$ are recently updated as 1.

Algorithm 1 Bitwise iBFS at Level $k+1$

```
1: $\text{BSA}_{k+1} \leftarrow \text{BSA}_k$
2: **forall** frontier $f$ **in parallel do**
3:     **foreach** neighbor $v$ of $f$ **do**
4:         **if** Top-Down **then**
5:             $\text{BSA}_{k+1}[\text{v}] = \text{BSA}_{k+1}[\text{v}]$ **OR_atomic** $\text{BSA}_k[\text{f}]$
6:         **else** // *Bottom-up*
7:             **if** $\text{BSA}_{k+1}[\text{f}]$==0xff...f **then**
8:                 break; // *Early termination*
9:             **end if**
10:            $\text{BSA}_{k+1}[\text{f}] = \text{BSA}_{k+1}[\text{f}]$ **OR** $\text{BSA}_k[\text{v}]$
11:        **end if**
12:    **end for**
13: **end for**
```

Using frontier 7 in Figure 2.24(a) as an example, vertex 7 is a frontier shared by the third and fourth BFS instances in Figure 2.3. During the inspection of vertex 7's neighbor – {5, 6, 8}, iBFS uses the bitwise `OR` operation between 7's status and each neighbor's status. Specifically, for vertex 5, it sets the third bit in $\text{BSA}_3[5]$, and for vertex 6, the fourth bit in $\text{BSA}_3[6]$. One may also notice that, the `OR` operation does not affect the already set bits, like the fourth bit of vertex 5 in $\text{BSA}_3[5]$. The reason is that the fourth BFS instance has already visited vertex 5 in prior levels.

Because multiple BFS instances may want to set different bits of the same vertex in the bitwise status array concurrently, iBFS needs atomic operations to avoid overwrites of the updates to BSA_{k+1}, another difference from [49].

Bottom-up traversal, similarly assigns a single thread to work on each frontier. However, the frontier's status is updated by the neighbors' statuses, that is, iBFS uses `OR` between $\text{BSA}_{k+1}[\text{f}]$ and $\text{BSA}_k[\text{v}]$ and stores the result into $\text{BSA}_{k+1}[\text{f}]$. Using frontier 7 of Figure 2.24(b) as an example, its neighbors are {5, 6, 8}. iBFS executes `OR` between vertex 5's and 7's statuses and stores the result in $\text{BSA}_4[7]$.

Early Termination. During bottom-up traversal for some frontiers, like frontier 7 in Figure 2.24(b), iBFS does not necessarily need to check all its neighbors because it is possible that some of its neighbors (e.g., vertex 5 for frontier 7) can set all bits of this frontier in the bitwise status array. When this happens, iBFS will terminate the

Algorithm 2 Frontier Identification at Level $k+1$

```
1: forall vertex $v$ in parallel do
2:     if Top-down then
3:         if $\text{BSA}_{k+1}[v]$ XOR $\text{BSA}_k[v]$ then
4:             JFQ.enqueue($v$)
5:         end if
6:     else // Bottom-up
7:         if NOT $\text{BSA}_{k+1}[v]$ then
8:             JFQ.enqueue($v$)
9:         end if
10:    end if
11: end for
```

inspection on this frontier, eliminating the need to further examine other neighbors (e.g., vertices 6 and 8), which we call early termination in this work. This newly freed-up thread will then be scheduled to work on other frontiers. This is not possible unless the bitwise operations already record all visited vertices as '1' in BSA, in this case before this level $\text{BSA}_4[7]$ already has three bits set. In short, the early termination allows a great reduction in the traversal time compared to prior work such as [49].

Bitwise Frontier Identification. iBFS also needs a different approach for frontier identification that works efficiently with the bitwise status array. Algorithm 2 shows the pseudocode for bitwise frontier identification.

Top-down traversal executes the `XOR` operation between BSA_k[v] and BSA_{k+1}[v]. If *true* is returned, it means some BFS instances have just changed the corresponding bit of v and stored in BSA_{k+1}[v]. Since top-down treats most recently visited vertices as frontiers, iBFS hence stores the identified frontiers in the joint frontier queue. In contrast, bottom-up traversal, is much easier since it treats unvisited vertices as frontiers. Therefore, iBFS simply performs a `NOT` operation on BSA_{k+1}[v]. If it is evaluated as *true*, that is, some bits of BSA_{k+1}[v] are not set, then this vertex is a frontier and iBFS puts it in the joint frontier queue.

Vector data types on GPUs are also leveraged in iBFS. Clearly, the number of bits in each variable affects the number of concurrent BFS, e.g., if BSA is implemented with `int` type, one variable can represent the statuses for 32 BFS instances. In CUDA, the basic data are `char`, `int` and `long`, and on the other hand `vector` packs multiple basic types into one, e.g., `int4` contains four `ints`, similarly `char4`, `long4` etc. Using the vector data type in iBFS can further reduce the memory access time as it fetches four basic data elements together at one time.

Summary. Our bit-wise approach is novel in a number of ways, which leads to upto 2.6× speedup over [49]. First, [49] resets the bitwise status array at each level and only records the current level frontiers as 1. In comparison, iBFS records every visited vertex as 1 regardless of which level it is visited. This way our bit-wise status array remembers all visited vertices, for which we also introduce a new frontier identification technique and early termination during bottom-up traversal as described earlier. Second, [49] is based on single thread BFS implementation, that is, each thread works on one BFS instance. In contrast, iBFS supports multi-threaded bitwise operation, that is, all available threads will work on a group of BFS instances. Inter-thread synchronization shall be carefully managed, because in iBFS multiple threads will need to update different bits of the same vertex simultaneously. In top-down, iBFS uses (manually controllable) shared memory on GPUs to cache and merge the updates in the same CTA (typically 256 threads), which avoids the overhead of atomic operations at this step. Next, iBFS has to rely on atomic operations to push the combined updates to the global memory. In bottom-up, iBFS performs multi-step tree-based merging of the updates within threads in a warp or CTA, again avoiding atomic operations.

2.6.4 iBFS on CPUs

In principal iBFS can be implemented on CPUs. Specifically, *joint traversal* and *GroupBy* can follow the same design on GPUs. One notable difference is that iBFS would need atomic operation on CPUs for the multi-thread *bitwise operation*. Note that [49] does not need atomic operations because it is based on single-thread BFS. Generally speaking, concurrent BFS is an I/O intensive application, as it always has many frontiers to be processed. As modern CPUs provide tens of cores and thousands of registers [133, 134], issuing a large number of CPU threads may improve memory

Name	Abbr.	Description	# Vertices (M)	# Edges (M)	Diameter	Directed
Facebook	FB	Facebook user to friend connection	16.8	421	10	Y
Friendster	FR	Friendster online social network	16.8	439.2	25	Y
Gowalla	GO	Gowalla online social network	0.2	1.9	–	N
Hollywood	HW	Hollywood movie actor network	1.1	115	10	N
Kron-20-512	KR0	Kronecker generator	1	1073.7	6	N
Kron-21-256	KR1	Kronecker generator	2.1	1073.7	7	N
Kron-22-128	KR2	Kronecker generator	4.2	1073.7	7	N
Kron-23-64	KR3	Kronecker generator	8.4	1073.7	7	N
Kron-24-32	KR4	Kronecker generator	16.8	1073.7	8	N
LiveJournal	LJ	LiveJournal online social network	4.8	69.4	15	N
Orkut	OR	Orkut online social network	3.1	234.4	9	N
Pokec	PK	Pokec online social network	1.6	30.1	11	Y
R-MAT	RM	GTgraph: R-mat generator	2	256	6	Y
Twitter	TW	Twitter follower connection	16.8	186.4	17	Y
Wikipedia	WK	Wikipedia page links in 2007	3.6	45	12	Y
Wiki-Talk	WT	Wikipedia talk network	2.4	5.0	–	Y
YouTube	YT	YouTube online social network	1.1	6.0	–	N

Table 2.1: Graph Specification

throughput but inevitably would incur high overhead of context switches. On the other hand, GPUs not only provide a large quantity of small cores coupled with huge register files, e.g., 2,880 cores and 983,040 registers on NVIDIA Kepler K40 GPUs, but also support zero-overhead context switch [9]. As we will present shortly, compared to the CPU-based implementation, GPU-based iBFS runs 2× faster on average on various graphs.

2.7 Enterprise Experiments

We have implemented Enterprise in 3,000 lines of code in C++ and CUDA. The source code is compiled with NVIDIA nvcc 5.5 and GCC 4.4.7 with the optimization flag of O3. In this work, we use three GPUs: NVIDIA Kepler K40, K20 and Fermi C2070. We perform our tests on the graphs described in Table 2.1. All the graphs are represented by compressed sparse row (CSR) format. The datasets that provide edge tuples are transformed into the CSR format, with the sequence of the edge tuples preserved. The majority of the graphs are sorted, e.g., Twitter and Facebook. We do not perform pre-processing such as removing duplicate edges or self-loops. All the data is represented by uint64 type, loaded into GPU's global memory. The timing starts when the search key is given to the GPU kernel and ends when the search is completed and written into the GPU memory. For each experiment, we run BFS 64 times on pseudo-randomly selected vertices and calculate the mean. The metric *traversed edges per second* (TEPS) is computed as follows: Let m be the number of directed edges traversed by the search, counting any multiple edges and self-loops, and t be the time elapsed during BFS search mentioned above. Then, TEPS is calculated by m/t.

2.7.1 Graph Benchmarks

We use a total of 17 graphs in this paper, as summarized in Table 2.1, which have vertices ranging from 1 to 17 million and edges from 30 million to over 1 billion. For an undirected graph, we count each edge as two directed edges. Eleven real world graphs are included such as Facebook [135], Twitter [7], Wikipedia [136], as well as the LiveJournal, Orkut, Friendster, Pokec, YouTube, Wiki-Talk and Gowalla social network graphs from the Stanford Large Network Dataset Collection [137]. In addition, we utilize two widely used graph generators, Kronecker [83] and Recursive MATrix (R-MAT) algorithm [69] [138]. Both generators take four possibilities A, B, C and $D = 1.0 - A - B - C$. The Kronecker generator produces the Kron-*Scale-EdgeFactor* graphs that have 2^{scale} number of vertices with the average out-degree of *EdgeFactor*. In this work, we use (A, B, C) of (0.57, 0.19, 0.19) for Kronecker, and (0.45, 0.15, 0.15) for R-MAT graphs. It is worthy to point out that both real-world and synthetic graphs exhibit small-world characteristics - as the majority of the vertices have small out-degree and account for the small percentage of the total number of edges, there exist a number of hub vertices with high out-degree.

2.7.2 Enterprise Performance

We implement direction-optimizing BFS with the status array approach as the baseline (BL) since atomic operation based frontier queue would be much slower. Here we use CTA to work on each vertex in the status array, which is much faster than assigning a thread or warp. Figure 2.25 plots the performance improvement contributed by each optimization including streamlined GPU threads scheduling (TS), GPU workload balancing (WB), hub vertex cache (HC).

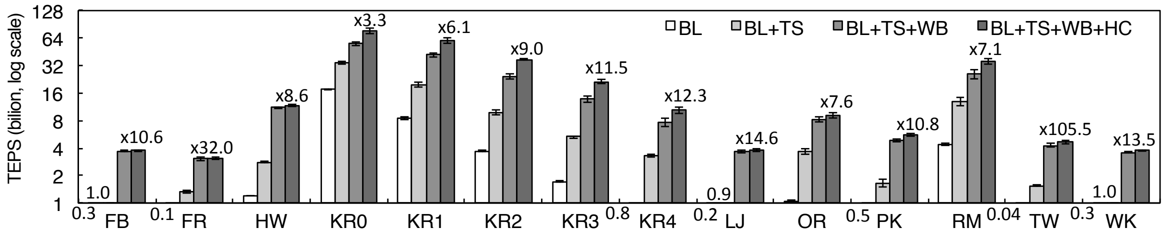


Figure 2.25: Enterprise performance on various graphs. Direction-optimizing BFS on GPU using the status array method serves as the baseline (**BL**). Three techniques are represented as **TS** for streamlined GPU **T**hreads **S**cheduling, **WB** for **W**orkload **B**alancing, and **HC** for **H**ub vertex **C**ache.

The streamlined GPU threads scheduling outperforms the baseline by 2× to 37.5× across all graphs. In particular, Twitter (TW) obtains the biggest speedup from 0.04 to 1.5 billion TEPS. The reason is that the maximum frontier ratio in Twitter is only 10.2%, and on average it only has 1% frontiers at each level. Kron-20-512 (KR0) gains 2× speedup, reaching 34 billion TEPS. In general, generating the frontier queue consumes on average 11% of the BFS run time.

The GPU workload balancing technique more than doubles the traversal rate for all graphs, 2.8× on average beyond the first technique. For example, LiveJournal (LJ) achieves the biggest improvement of 4.1×, from 0.9 to 3.7 billion TEPS. For this graph, the total workload is distributed evenly so that SmallQueue contains 78% frontiers (or 22% workload), MiddleQueue has 21% frontiers (or 58% workload), LargeQueue 1% frontiers (20% workload).

The hub vertex caching technique helps improve the performance up to 55%. Both Facebook (FB) and Friendster (FR) see a small gain as they do not contain vertices with extremely high out-degree, e.g., the maximum out-degree in Facebook is 9,170. For other graphs, the improvement is more than 10%, as high as 30% to 50% for Kronecker graphs that have thousands of vertices with more than 10^5 edges. This shows that caching these hub vertices is very beneficial.

In all, Enterprise improves the TEPS of the BFS algorithm by 3.3× to 105.5×. The highest TEPS is achieved at KR0 with over 76 billion TEPS and the smallest at FR with 3.1 billion TEPS.

Comparison. Figure 2.35 compares Enterprise with several GPU based BFS imple-

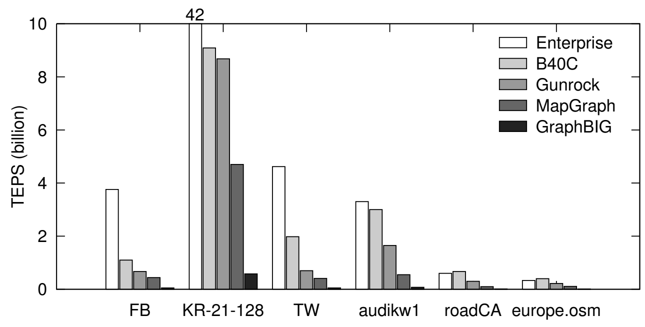


Figure 2.26: Performance comparison.

mentations, including B40C [74], Gunrock [75], MapGraph [139] and GraphBIG [140]. We evaluate power-law graphs such as FB, TW, and KR-21-128 which has 2 million vertices with average out-degree of 128, as well as high-diameter graphs such as audikw1 [136], roadCA [137] and europe.osm [136].

For power-law graphs, Enterprise performs 4×, 5×, 9× and 74× better than B40C, Gunrock, MapGraph and GraphBIG, respectively. For high diameter graphs, Enterprise achieves 1.41 billion TEPS on average and outperforms Gunrock (0.72) 1.95×, MapGraph (0.25) 5.56×, GraphBIG (0.03) 42×. On these graphs, Enterprise delivers similar perform as B40C. It runs slightly slower on europe.osm because this graph has very small out-degrees, with the maximum out-degree of 12 and the mean 2.1.

2.7.3 Enterprise Scalability

Figure 2.32 shows both strong and weak scalability of Enterprise. We use the largest graph from Table 2.1, i.e., KR4 to test the strong scalability. On 2, 4 and 8 GPUs, Enterprise achieves 15, 18 and 18.4 billion TEPS, respectively, that is, a speedup of 43%, 71% and 75%.

We evaluate weak scalability in two ways, edge scale and vertex scale. When the GPU count increases, we increase the edgeFactor – the average out-degree – with fixed vertex count, or increase the number of vertices with the constant edgeFactor. As shown in Figure 2.32, we achieve better scalability for edge scale, where we obtain super linear speedup, that is, 9.1×, 96 billion TEPS with 8 GPU. This is because

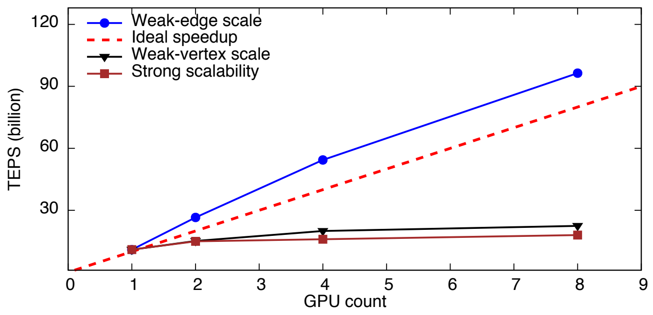


Figure 2.27: Strong and weak scalability of Enterprise.

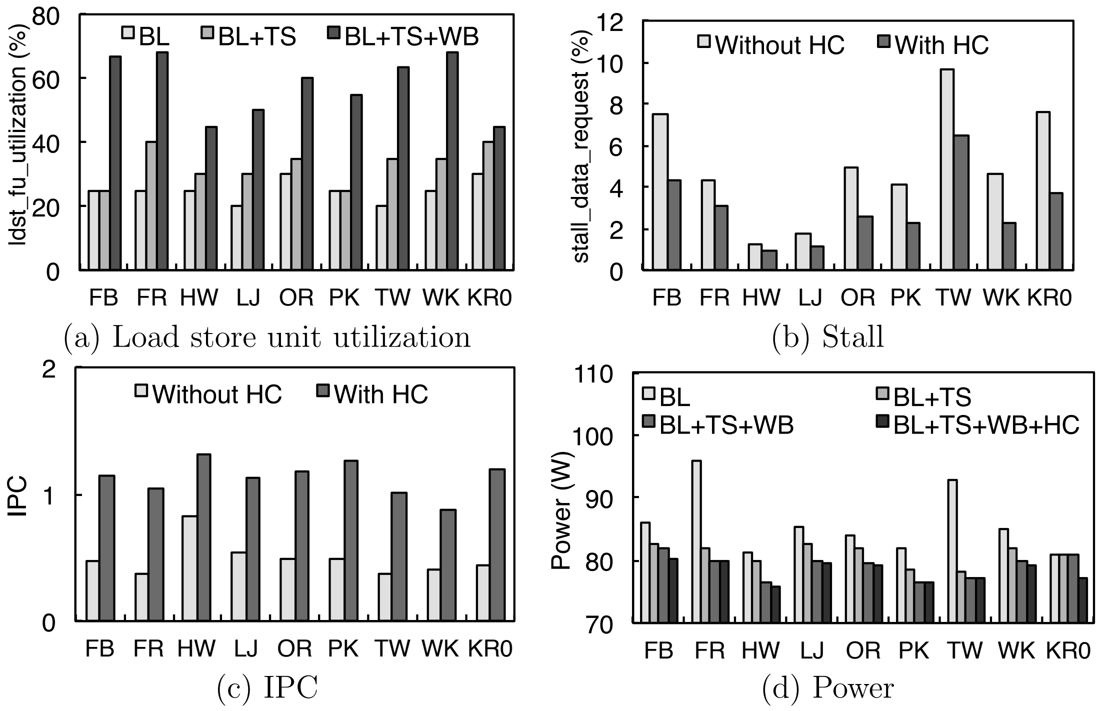


Figure 2.28: Microarchitecture profiling statistics of Enterprise: (a) Load/store function unit utilization (b) Stall caused by data request (c) IPC (d) GPU power consumption.

when edgeFactor increases, the number of hub vertices increases in the graph too and the hub vertex cache will reduce more global memory accesses. On the other hand, direction-switching can possibly avoid more unnecessary edge checks.

2.7.4 Analysis of GPU Counters

As BFS is an I/O-intensive algorithm, it is critical that GPU threads are able to access data quickly. As shown in Figure 2.28(a), our frontier techniques (TS and WB) significantly improve the utilization of GPU load/store function units by average 8% and 24% respectively, reaching as high as 68%. Furthermore, our hub vertex caching (HC) presented in Figure 2.28(b), reduces the stalls of data requests by 40%, the occurring of such events drops from 4.8% to 2.9%. This also explains the double of IPC observed on GPUs in Figure 2.28(c).

For comparison, we also profile [74] on Hollywood graph, which delivers 2.7 billion TEPS while consumes 40 Watts power, achieves 40% utilization of load/store unit and 0.68 IPC. On the same graph, Enterprise achieves 50% load/store unit utilization and 1.32 IPC, with 12 billion TEPS and 76 Watts power consumption.

Figure 2.28(d) plots GPU's power consumption corresponding to different techniques. Here we only report GPU's power to understand the impact of each technique. On average, the power consumption drops from 86 to 81 Watts with our GPU threads scheduling, the biggest saving of 14.5 Watts on the Twitter graph. This comes mostly from better IO performance and fewer idle GPU threads in the system. The other two techniques (WB and HC) further reduce the power to 78 Watts.

2.8 iBFS Experiments

iBFS is implemented in 4,000 lines of CUDA and C++ codes, extending a GPU-based high-performance BFS implementation – Enterprise [1]. Since it supports both top-down and bottom-up BFS, our iBFS can be easily configured to support conventional top-down BFS and traverse weighted graphs. iBFS is compiled using g++ 4.4.7, MPI (MVAPICH2) and NVIDIA CUDA 6.0 with the *-O3* flag. Our system is evaluated on NVIDIA Kepler K40 GPUs on our local cluster with Intel Xeon E5-2683 CPUs, and later on the Stampede supercomputer on NSF Extreme Science and Engineering Discovery Environment (XSEDE) program, where iBFS runs from 1 to 112 machines each of which is equipped with one NVIDIA Kepler K20 GPU.

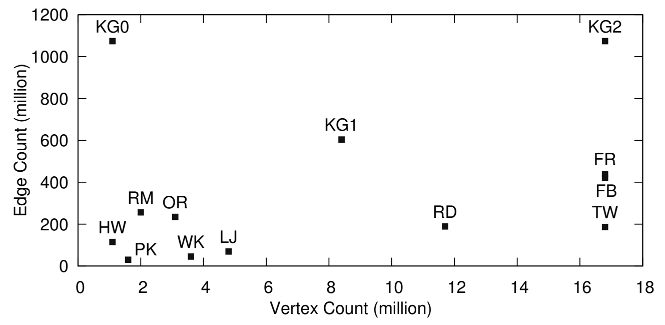


Figure 2.29: Graph benchmarks.

We measure the execution time from when all the data are loaded in GPU memory to the traversal is completed and the results are stored in GPU memory. All the execution uses uint64 data type. We use the metric of *traversed edges per second* (TEPS) to measure the performance, which is calculated by the ratio of the number of directed edges traversed by the search, counting any multiple edges and self-loops, and the time elapsed during iBFS execution. In the tests, iBFS performs breadth-first search from all the vertices.

2.8.1 Graph Benchmarks

In this work we use total 13 graph benchmarks to evaluate iBFS, which as summarized in Figure 2.29 have upto 17 million vertices and 1 billion edges. In particular, there are seven real-world graphs of well-known online social networks. Facebook (FB) [135], a user to friend connection graph, contains 16,777,216 vertices and 420,914,604 edges. Twitter (TW) [7] is a follower connection graph, that is, if user v follows user u, (v, u) is considered as an edge. It also has 16,777,216 vertices and 196,427,854 different edges. Wikipedia (WK) [136] is a inter-website hyper-link graph, which consists of 3,566,908 vertices and 45,030,389 edges. We also obtain four popular online social network graphs, i.e., LiveJournal (LJ), Orkut (OR), Friendster (FR), and Pokec (PK), from Stanford Large Network Dataset Collection [137]. Specifically, LJ contains 4,847,571 vertices and 137,987,546 edges. OR has 3,072,627 vertices with an average outdegree of 75.27. FR contains 16,777,212 vertices and 439,147,122 edges. PK is the smallest graph with 1,632,804 vertices and 30,622,564 edges.

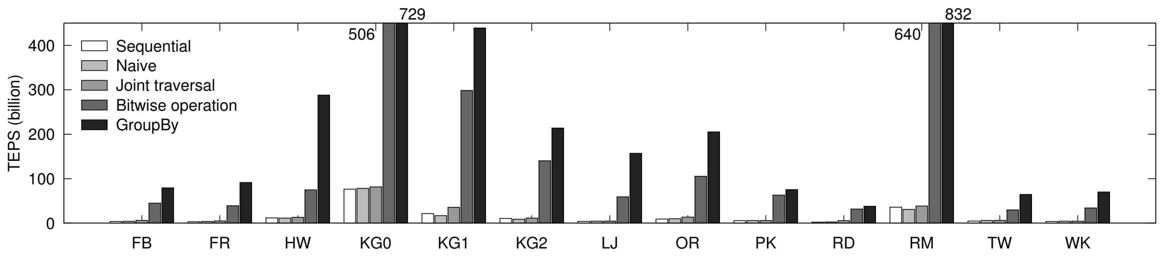


Figure 2.30: Traversal rate comparison between Sequential BFS, Concurrent BFS, Joint Traversal, Bitwise Optimization, and GroupBy.

In addition, we generate three types of synthetic graphs with Graph 500 generator [69, 83, 65], namely, KG0, KG1 and KG2. The default value of (A, B, C) parameter is (0.57, 0.19, 0.19) per the requirement of Graph 500 [83]. Specifically, KG0 stands for the high average outdegree graphs, i.e., its average outdegree is 1024 and vertex count is 1,048,576. KG2 serves as the biggest graph in this paper, i.e., with both biggest vertex and edge count – 16,777,216 vertices and 1,073,741,824 edges. KG1 has 8,388,608 vertices and 603,979,776 edges. We also use the DIMACS graph generator [138] to generate the RM and RD graphs in this paper. RM follows the same theory from Graph 500 [69, 83, 65] but with a different (A, B, C) set of (0.45, 0.15, 0.15). RM has 2,097,152 vertices and 268,435,456 edges. RD [141] graph has uniform outdegree distribution, i.e., each vertex has roughly the same outdegree. RD contains 11,796,480 vertices and 188,743,680 edges.

All these graphs are stored in the *Compressed Sparse Row* (CSR) format. For graphs that are provided in the edge list format, we translate them into CSR while preserving the edge sequence. For undirected graphs, each edge is considered as two directed edges. For directed graphs, we also store the reversed edges to support the bottom-up traversal. The size of these graph ranges from 478 MB (PK) to 8.2 GB (KG2) when using *long integer* (8 bytes) to represent the vertex id.

2.8.2 iBFS Performance

We evaluate three techniques, *joint traversal*, *bitwise operation*, and *GroupBy*, and compare against running all BFS instances sequentially (*sequential*) or in parallel without any optimization (*naive*), both of which are based on state-of-the-art BFS

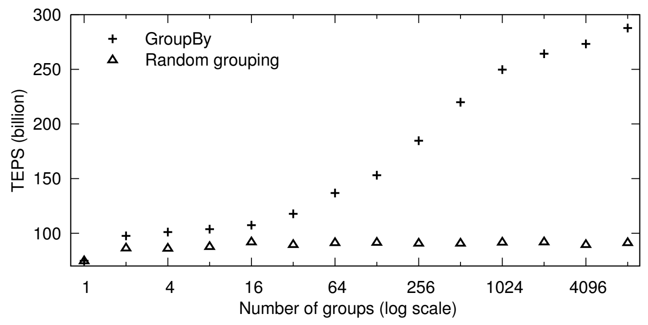


Figure 2.31: Traversal performance when running different number of BFS groups on HW.

implementation such as Enterprise [1]. In this test, we run APSP on all the graphs. As shown in Figure 2.30, sequential and naive implementation perform roughly the same. On average, the later traverses 1.05× faster, but a worse performance is observed in HW, KG1, KG2 and RM graphs, with only 78% of sequential performance on KG1.

In iBFS, joint traversal delivers 1.4× speedup compared to sequential implementation. The biggest performance gain of 2.6× speedup comes for RD, from 4.4 to 11.3 billion TEPS while the smallest is 1.03× speedup for PK . For other graphs, the performance improvement is more than 10%. On the other hand, bitwise iBFS, on average speedup the traversal by 11×. While the smallest improvement is 6.4× on HW from 11.7 to 75 billion TEPS, the biggest improvement is observed on RM – 18 × speedup from 36 to 640 billion TEPS. Astoundingly, **Groupby** improves iBFS by additional 2× on average beyond bitwise optimization, with the highest traversal rate of 832 billion TEPS on RM.

In this test, iBFS greatly reduces the run time of APSP on all graphs from average 123 hours of sequential traversal to 5.5 hours, where joint traversal helps to reduce to 87.8 hours, bitwise optimization 11.2 hours, and GroupBy the rest. GroupBy incurs a minimal processing time of less than 0.1 second on average.

We further evaluate the traversal performance of iBFS as the number of i varies, that is, running MSSP with varied number of source vertices. We observe similar performance on all the graph benchmarks. Figure 2.31 presents the TEPS when running different number of BFS groups on the HW graph, where the total number of BFS instances equals to the multiply of the number of groups and the group size.

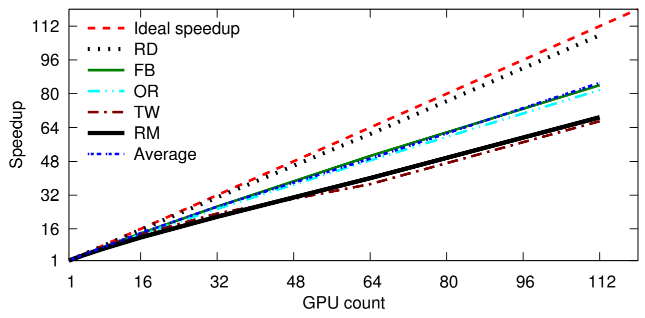


Figure 2.32: Scalability of bitwise iBFS from 1 to 112 GPUs.

Clearly, as there are more BFS instances to run, the benefit of GroupBy – the gap between GroupBy and random grouping – increases because better groups can be formed. Specifically, with randomly formed groups, the traversal rate fluctuates in the range of 75 and 90 billion TEPS, which is raised to 288 billion TEPS with the help of GroupBy.

2.8.3 iBFS Scalability

In this test, we aim to evaluate the scalability of iBFS on a large number of distributed GPUs on the Stampede supercomputer at TACC. Clearly, as long as different GPUs work on independent BFSes, there is no need for inter-GPU communication. Therefore, the key challenge here is achieving workload balance on GPUs, especially when each individual BFS may inspect different number of edges during bottom-up. The longest time consumption of all the GPUs is reported in this test.

For the five graphs tested, iBFS achieves good speedup from 1 to 112 GPUs (the total number of GPUs on Stampede)[2]. As shown in Figure 2.32, from one to two GPUs, the biggest speedup, as expected, is from RD of 1.97× because RD graph has the most balanced workload. And even the smallest speedup from OR is 1.9×. From one to four GPUs, the average speedup is 3.8× with the smallest speedup from OR of 3.6× while the biggest from RD graph – 3.9×. As the GPU count increases, workload imbalance slowly emerges and begins to negatively affect the performance. Specifically, iBFS achieves an average of 85× speedup for 112 GPUs. Again, RD gets

[2]As it took all the GPUs on Stampede, we were only given a small time window to conduct this test.

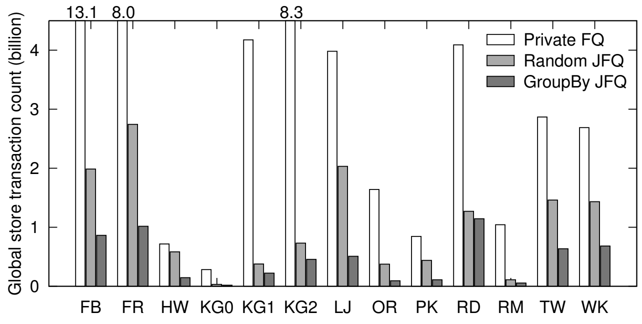


Figure 2.33: Global store transaction count during the generation of private frontier queue, random joint frontier queue and GroupBy joint frontier queue.

the biggest speedup, i.e., 108×.

In all, iBFS achieves the average traversal rate of 16,509 billion TEPS across all the tested graphs with the maximum 57,267 billion TEPS on RM.

2.8.4 Joint Traversal and GroupBy

This section uses the NVIDIA profiler [64] to measure the impact of joint traversal and GroupBy on memory accesses. Having a joint frontier queue with only one copy of shared frontiers reduces the potentially large number of global memory writes, compared to having private frontier queues for each BFS instance. Figure 2.33 shows the number of global store transactions during the frontier queue generation of 1,024 BFS instances. Using private frontier queue, i.e., *private FQ*, executes 4 billion transactions on average across all the graphs, while joint frontier queue with random groups, i.e. *random FQ*, only needs one fourth transactions. The biggest reduction of over 11× is observed in the KG2 graph, from 8.3 billion to 700 million, the smallest saving is 1.2× on HW, i.e., 720 million to 580 million. With *GroupBy*, iBFS further saves the global stores by 2.6× with the largest from HW of 4×.

Combining joint status array with careful thread scheduling introduces significant performance benefits, as iBFS minimizes costly memory operations. Figure 2.34 exhibits global load transactions per request during traversal before and after this optimization. Note that global store transactions per request exhibits similar trend. Since joint status array of iBFS always coalesces the inspections and updates from

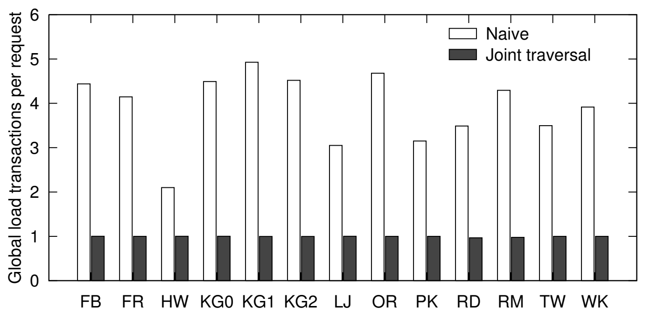


Figure 2.34: Load transaction count per request.

contiguous threads into a single global memory transaction, our tests across 1,024 BFS instances show that we are able to reduce on average from four loads to a single load. The benefits add up quickly considering a large number of memory operations in concurrent BFS. On average each BFS instance executes 50 million of global load transactions.

2.8.5 Bitwise Operation

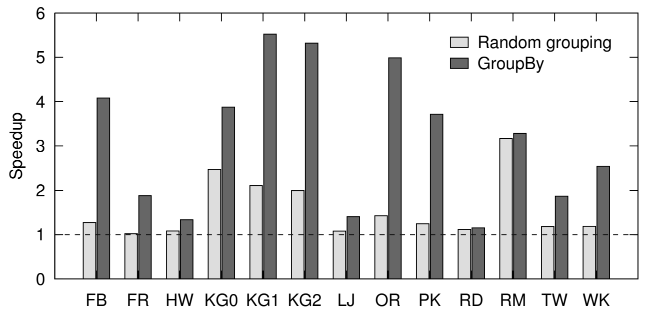


Figure 2.35: The speedup of our bitwise operation.

We implement the bitwise operation as in [49] and use it as baseline running all the benchmarks in our paper. Figure 2.35 plots the speedup of our bitwise operation. Even with random groups, we achieve 40% speedup, and our bitwise operation enjoys a better speedup with the outdegree-based GroupBy rules, i.e., 2.6×. Specifically, in random grouping, the maximum and minimum speedup is 2.5× of KG2 and 2% of FR. In comparison, the maximum and minimum speedup in GroupBy is 5.5× of KG1 and 15% of RD graph. Additional improvement from Groupby comes from the combined effect of high sharing ratio across grouped BFS instances and early termination enabled by bitwise traversal, which as a result allows concurrent instances to complete together and as early as possible.

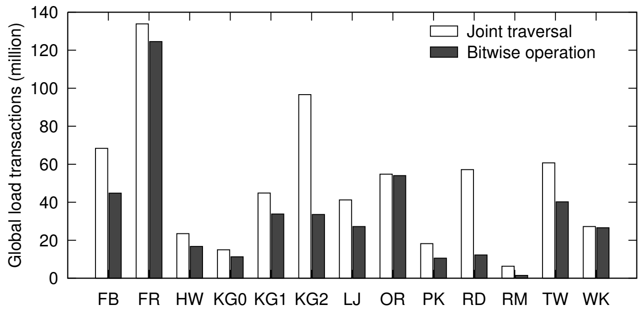


Figure 2.36: Total number of load transactions.

In addition, we evaluate the impacts on the total number of global load transactions by bitwise traversal. Because bitwise status array consolidates the statuses of multiple (128 in our case) vertices into a single variable, we reduce the global load transactions of 1,024 BFS instances by 40%, i.e., from 53 to 38 million on average presented in Figure 2.36.

2.8.6 Comparison of State of the Art

We have implemented two CPU-based concurrent BFS, namely MS-BFS [49] and our own iBFS, both of which run 64 threads in total. In addition, we compare our GPU-based iBFS with B40C [74] and SpMM-BC [82]. B40C runs a single BFS instance on GPUs and has similar performance as the sequential or naive implementation presented in Figure 2.30, and SpMM-BC uses a simple GPU-based concurrent BFS to calculate betweenness centrality. Figure 2.37 presents the performance of all five implementations when running APSP on six different graphs.

CPU-based iBFS is significantly faster than MS-BFS thanks to the techniques such as GroupBy and early termination. The biggest speedup is achieved for the KG0 graph, where MS-BFS obtains 120 billion TEPS while our CPU-based iBFS reaches 397 billion TEPS. iBFS also achieves an average improvement of 45% on other five graphs. On GPUs, iBFS traverses on average 2× faster than SpMM-BC, and 19.3× than B40C. For graphs KG0 and HW, iBFS delivers about 700 and 300

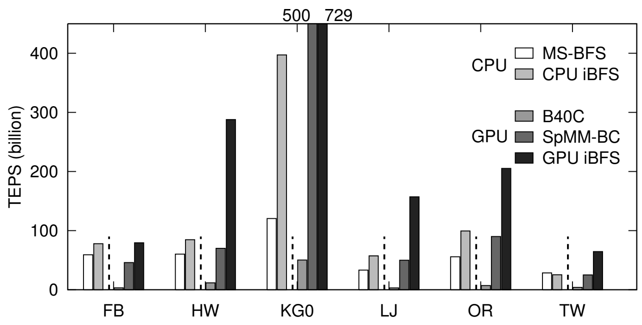


Figure 2.37: Comparison of CPU and GPU implementations.

TEPS respectively, greatly outperforming the other two implementations. Compared to the CPU-based implementation, GPU-based iBFS runs 2× faster on average across six different graphs.

Dataset	CPU		GPU	
	MS-BFS	CPU-iBFS	B40C	GPU-iBFS
FB	19.2	16.5	302.8	14.3
KG0	1.85	0.56	2.9	0.31
OR	4.1	3.22	25.3	1.1
TW	2.1	2.7	27.8	0.9

Table 2.2: Runtime (hours) of 3-hop reachability index.

2.8.7 Application: Reachability Index

To illustrate the broader application of iBFS on graph algorithms, in this paper we evaluate the benefits of using iBFS to construct the index for answering graph reachability queries, which computes the first k levels BFS for a large amount of selected vertices. Table 2.2 lists the runtimes of various implementations for constructing the index for 3-hop reachability. Clearly, GPU-based iBFS outperforms other concurrent BFS systems on four different graphs. Specifically, it is 21×, 3.3× and 2.2× faster than B40C, MS-BFS and CPU-based iBFS, respectively.

Chapter 3

SIMD-X Programming and Processing of Graph Algorithms on GPUs

3.1 Introduction

The advent of big data represents both a challenge and opportunity, from which extracting useful knowledge within an acceptable time envelope remains elusive. Many applications leverage graphics processing units (GPUs) for performance acceleration, where huge success comes from exploiting data-level parallelism in these applications, that is, the *single instruction multiple data* (SIMD) model of GPUs.

Implicitly, the traditional SIMD model assumes *regular* programming and processing, where not only the same instruction is executed but also the same amount of work is performed on each piece of data. Unfortunately, this assumption is no longer valid for many emerging *irregular applications*, especially graph analytics the focus of this work. Such applications do not conform to the SIMD model, where different amount of work, or completely different work, need to be performed on the data in parallel.

To enable accelerated graph computation on GPUs, this work advocates a new

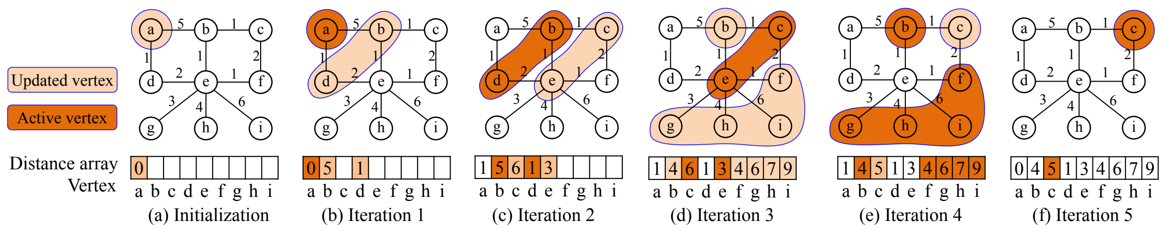


Figure 3.1: SSSP on a graph, with nine vertices {a, b, c, d, e, f, g, h, i} and ten undirected edges (with weights). SSSP iteratively computes on the graph and generates the distance array. Particularly, heavy and light shadows represent active and most recently updated vertices, respectively.

parallel framework, *SIMD-X*, for the programming and processing of *single instruction multiple,* ***complex****, data* on GPUs. At the heart of SIMD-X is the decoupling of graph programming and processing, that is, SIMD-X utilizes the *data-parallel* model for ease of programming of graph applications, while enabling system-level optimizations to deal with *task-parallel* complexity at run time.

With SIMD-X, a programmer simply needs to define what to do on which data, without worrying about issues arisen from irregular memory access and control flow. Yet, SIMD-X is still able to deliver exceptional performance, e.g., 7× faster than Gunrock [75] with 5× less lines of code (LOC). SIMD-X consists of three major components:

First, SIMD-X utilizes a new *Active-Compute-Combine* (ACC) programming model that asks a program to define three data-parallel functions: the condition for determining an active vertex, computation to be performed on an associated edge, and combining the updates. SIMD-X is slightly different from the traditional edge-centric model, that is, instead of immediately updating vertex status with an atomic operation, the framework takes care of the updates and applies performance optimizations as in vertex-centric processing. As we will show later, ACC is able to support a large variety of graph algorithms from breadth-first search, k-core, to belief propagation.

Second, SIMD-X relies on just-in-time (JIT) task management which is able to balance parallel workloads across different GPU cores, with minimal overhead. A good task list can increase not only parallelism, but also sequential memory access for the computation of next iteration, both of which are crucial for high-performance computing on GPUs. To this end, we have designed new online and ballot filters, each of which excels at the complementary scenarios, i.e., the former favors a small amount of tasks while the latter larger tasks. At runtime, SIMD-X judiciously selects the more suitable filter to assemble the active work list for the next iteration. As a result, SIMD-X is able to deliver up to two orders of magnitude speedup (average 16×) across various algorithms and graphs.

Third, SIMD-X designs another new technique of push-pull based kernel fusion which aims to further accelerate graph computing by reducing kernel invocation overhead and global memory traffic. Instead of aggressively fusing the algorithm into one giant kernel, SIMD-X fuses the kernels around the pull and push stages within each computation. The evaluation shows that the new fusion technique can reduce the register consumption by half and thus double the configurable thread count, leading to 42% and 25% performance improvement over non-fused and aggressive fusion, respectively.

SIMD-X is different from prior work in several aspects. First, in order to use existing systems efficiently, a programmer needs to possess an in-depth knowledge of GPU architecture [142, 9], e.g., Gunrock requires explicit management of GPU threads and memory [75], and B40C [74] and Enterprise [1] contains thousands of lines of CUDA code for BFS specific optimizations. One of the goals of this work is to provide a simple programming model and delegate the responsibility of task management to SIMD-X. Second, current systems either ignore workload imbalance as in [143, 120], or resolve it reactively as in [75, 109], both of which result in undesired system performance. Lastly, because GPUs lack support for global synchronization, existing systems [144, 75, 1, 2] rely on the multi-kernel design, which comes with considerable overhead, especially for graph algorithms with high iteration count. SIMD-X addresses these challenges with the help of new filters, as well as a deadlock-free software barrier.

The rest of this chapter is organized as follows: Section 3.2 discusses the related work. Section 3.3 presents the challenges of constructing SIMD-X on GPUs. Section 3.4 describes the ACC model. Section 3.5 presents our just-in-time management

GPU systems	LOC	Task management	Kernel fusion
Luo et al. [106]	2,000	Atomic filter	No
B40C [74]	5,000	Batch filter	No
CuSha [143]	200	-	No
Enterprise [1]	3,000	Strided filter	No
Gunrock [75]	600	Batch filter	No
SIMD-X	80	Just-in-time control	Push-pull based

Table 3.1: Summary of related work.

approach and Section 3.6 discuses the kernel fusion design. We present the graph algorithms in Section 3.7 and the evaluation results in Section 3.8.

3.2 Related Work

Recent advance in graph computing falls broadly in algorithm design innovation [60, 72], framework developments [145, 112, 146, 147, 5, 76, 148, 149, 71, 150, 151, 152, 6, 153, 154] and accelerator based optimizations [155, 156, 157, 158, 1, 74, 143]. In this work, we discuss relevant work from three aspects – programming model, task management and kernel fusion. Table 3.1 compares five representative projects with SIMD-X.

Besides edge and vertex centric models, there are also other models that make various trade-offs between simplicity and performance. For instance, "think like a graph" [159] requires each vertex to obtain the view of the entire partition on one machine in order to minimize the communication cost. Furthermore, domain specific programming language systems, such as Galois [6], Green-Marl [152] and Trinity [150], allow programmers to write single-threaded source code while enjoying multi-threaded processing. In comparison, SIMD-X decouples the goal of programming simplicity and performance: with ACC, SIMD-X ultimately designs a data-parallel abstraction for deploying irregular graph applications on GPU. With JIT task management and push-pull based kernel fusion, SIMD-X pushes the performance towards a magnitude faster than state-of-the-art CPU and GPU frameworks.

Task management is an important optimization for GPU-based graph computing.

Besides batch filter [75, 74], there also exist other task management approaches – strided filter [1, 2] and atomic filter [106]. Particularly, strided filter resembles ballot filter but the former one experiences strided memory access when scanning the metadata thus performs up to 16× worse than ballot filter. Atomic filter relies is similar to online filter but it relies on automic operation to put active vertices into global active list which suffers from orders of magnitude slow down than online filter. Besides ballot and online filter bests batch, stride and atomic filter, SIMD-X goes further via introducing a JIT controller to dynamically use online filter and ballot filter and further improve the performance.

Kernel fusion affects applications far beyond graph computations. SIMD-X demonstrates its benefits in graph computing and machine learning (MLP) applications. SIMD-X is closely related to global software barrier [160, 161]. However, previous work fails to identify the deadlock issue in this global software barrier problem, thus no solution towards this issue. In contrast, SIMD-X unveils, systematically analyzes, and resolves this problem. To avoid high register consumption, SIMD-X further selectively fuse kernels via exploiting the special kernel launching patterns of graph algorithms. It is also important to mention existing work [144] that only fuse kernels to barrier boundary. In comparison, SIMD-X fuses kernels across barriers.

3.3 SIMD-X Challenges and Architecture

3.3.1 Graph Computing on GPUs

Generally speaking, regular applications present uniform workload distribution across the data set. As a result, such applications lend themselves to the data-parallel GPU architecture. For development and evaluation, this work mainly uses NVIDIA GPUs, which have tens of streaming processors and in total thousands of Compute Unified Device Architecture (CUDA) cores [9, 63]. Typically, a *warp* of 32 threads execute the same instruction in parallel on consecutive data. For regular application, programming and processing is simple, e.g., dense matrix algebra as shown in Figure 3.2(a).

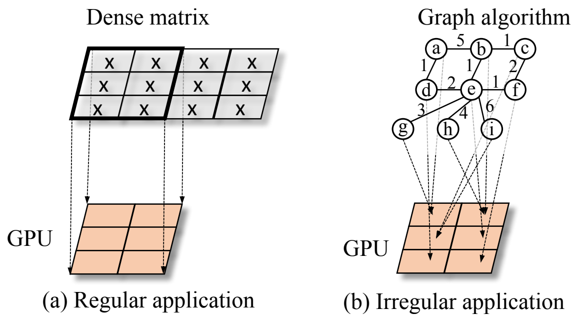


Figure 3.2: Mapping regular versus irregular applications on GPUs

On the other hand, task management for irregular applications is challenging on GPUs. In this work, we focus on a number of graph algorithms such as breadth-first search, k-core, and belief propagation. Here we use one algorithm – Single Source Shortest Path (SSSP) – to illustrates the challenges. Simply put, a graph algorithm computes on a graph $G = (V,\ E,\ w)$, where V, E and w are the sets of vertices, edges, and edge weights. The computation updates the algorithmic metadata which are the states of vertices or edges in an iterative manner. A typical workflow of SSSP is shown in Figure 3.1. Initially, SSSP assigns the infinite distance to each vertex in the *distance array*, which is represented as blank in the figure. Assuming the source vertex is a, the algorithm assigns 0 as its initial distance, and now vertex a becomes active. Next, SSSP computes on this vertex, that is, calculating the updates for all the neighbors of vertex a. In this case, vertices $\{b,\ d\}$ have their distances updated to 5 and 1 in the distance array. At the next iteration, the vertices with newly updated distances become active and perform the same computation again. This process continues until no vertex gets updated. Different from breadth-first search, SSSP may update the distances of some vertices across multiple iterations, e.g., vertex b is updated in iteration 1 and 3.

In this example, not every vertex is active at all time, and vertices with different degrees (number of edges) yield varying amounts of workloads. For instance, at iteration 3 of Figure 3.1(d), one thread working on vertex c computes two neighbors, while another thread on vertex e four neighbors. As a result, a complex mapping as shown in Figure 3.2(b) is required for high-performance processing, and to do so necessitates in-depth knowledge from a programmer on GPUs.

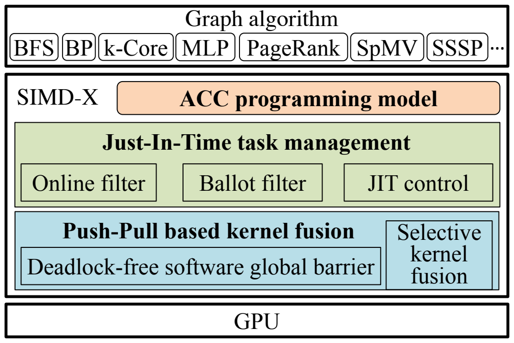
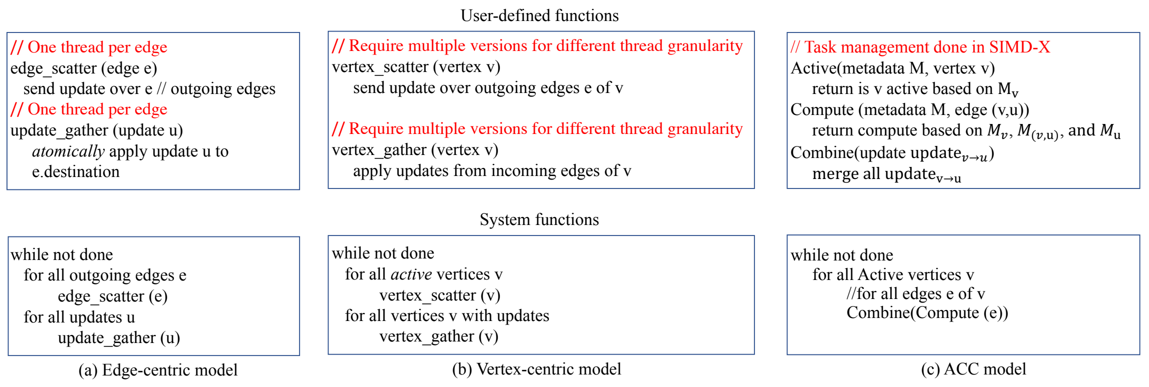


Figure 3.3: SIMD-X architecture

3.3.2 Architecture

SIMD-X is motivated to achieve two goals simultaneously: providing ease of programming for a large variety of graph algorithms, whereas enabling fine-grained optimization of GPU resources at the runtime. Figure 3.3 presents an overview of SIMD-X architecture. To achieve the first goal, SIMD-X utilizes a simple yet powerful Active-Compute-Combine (ACC) model. This data-parallel API allows a programmer to implement graph algorithms with tens of lines of code (LOC). Prior work requires significant programming effort [74, 1, 75], or runs the risk of poor performance [143].

In SIMD-X, high-performance graph processing on GPUs is achieved through the development of two components: (1) JIT task management, which is responsible for translating data-parallel code to parallel tasks on GPUs. Essentially, SIMD-X "filters" the inactive tasks and groups similar ones to run on the underlying SIMD architecture. In particular, SIMD-X develops online and ballot filters for handling different types of tasks, and dynamically selects the better filter during the execution of the algorithm. And (2) Pull-push based kernel fusion. Graph applications are iterative in nature and thus require synchronizations. Fusing kernels across iterations would yield indispensable benefits, because kernel launching at each iteration incurs non-trivial overhead. In SIMD-X, we observe that with aggressive kernel fusion, register consumption would increase dramatically, lowering the occupancy and thus performance. To this end, SIMD-X deploys kernel fusion around pull and push stages of each graph computation, seeking a sweet spot that not only maximizes the range of each kernel fusion but also minimizes the register consumption. It is worthy noting that we also address the deadlock issue faced by software global barrier in SIMD-X.



```
// One thread per edge
edge_scatter (edge e)
  send update over e // outgoing edges
// One thread per edge
update_gather (update u)
    atomically apply update u to
    e.destination
```

```
// Require multiple versions for different thread granularity
vertex_scatter (vertex v)
    send update over outgoing edges e of v

// Require multiple versions for different thread granularity
vertex_gather (vertex v)
    apply updates from incoming edges of v
```

```
// Task management done in SIMD-X
Active(metadata M, vertex v)
    return is v active based on $M_v$
Compute (metadata M, edge (v,u))
    return compute based on $M_v$, $M_{(v,u)}$, and $M_u$
Combine(update update$_{v\to u}$)
    merge all update$_{v\to u}$
```

```
while not done
  for all outgoing edges e
        edge_scatter (e)
  for all updates u
        update_gather (u)
```

```
while not done
  for all active vertices v
        vertex_scatter (v)
  for all vertices v with updates
        vertex_gather (v)
```

```
while not done
    for all Active vertices v
          //for all edges e of v
          Combine(Compute (e))
```

Figure 3.4: Different Programming Models

3.4 ACC Programming Model

When it comes to graph computing, there are two main programming models: vertex-centric vs. edge-centric. "Think like a vertex" [145, 76] focuses on tasks on active vertices in a graph, whereas "think like an edge" [151, 153] iterates on edges and simplifies programming. SIMD-X aims to achieve the dual goal of ease of programming in edge-centric model, and efficient workload scheduling in vertex-centric model.

3.4.1 Background

Edge-centric model is initially introduced by external-memory graph engine X-stream [151] to improve IO performance. It requires a programmer to define two functions needed on each edge, edge_scatter and edge_gather. As such, this model schedules threads by the edge count. Figure 3.4(a) shows SSSP with such a model. Particularly, one thread needs to send the distance value of the source vertex and the weight of the outbound edge to the destination vertex (edge_scatter), which atomically applies the new updates in edge_gather.

Edge-centric processing is a good fit for GPUs, because its data-level parallelism matches the SIMD. Importantly, this model allows the graph processing system like SIMD-X to handle additional functions, e.g., scheduling the tasks, and launching the kernels. For this reason, SIMD-X extends edge-centric model to leverage this opportunity.

However, edge-centric model, albeit a straightforward mapping, would not run efficiently on GPUs right out of the box. For one, the scatter function requires the atomic operation to avoid the write-after-write data hazard across different threads, which is understandably expensive on GPUs. More critically, since not every edge is active at each iteration, assigning one GPU thread per edge would lead to significant waste of GPU resources. To address this problem, one shall be able to identify active edges at runtime and dispatch GPU threads to work only on them. However, for graph algorithms, identifying active vertices would come more naturally, as we will see shortly.

Vertex-centric model focuses on the calculation to be done on the vertices, instead of the edges. Similarly, this model presented in Figure 3.4(b) contains two functions, vertex_scatter that defines what operations should be done on this vertex, and vertex_gather that applies the updates on the vertex. This model has been adopted by a number of existing projects, e.g., Pregel [145], GraphLab [146], PowerGraph [112], GraphChi [147], FlashGraph [76], and GridGraph [148], as well as GPU-based implementation such as CuSha [143] and Gunrock [75].

While two models are conceptually equivalent, their performance vary drastically in practice. The advantage enjoyed by vertex-centric model can be attributed to two techniques. First, the vertex_gather function can avoid locks by first combining the updates from the incoming edges. Second, the systems using this model require sophisticated task management to create a work lists of active vertices (*active list*), as well as to balance the workloads across different groups of GPU threads (e.g., warp, CTA, or Grid). The complexity of such technique is evidenced by the high LOCs in many systems presented in Table 3.1. It is important to note that current systems cannot hide this complexity from programmers, because one has to schedule GPU threads, statically or dynamically, within the verex_scatter and verex_gather functions, leading to multiple versions of the same function for different thread granularity.

In contrast, the ACC model in SIMD-X combines the advantages of both models,

```
Active (vertex v){
    return true
}

Compute (edge e, weight w){
    vect_value = metadata_prev[e.dest];
    return vect_val * w;
}

Combine (metadata_t *A){
    return sum(A);
}
```

(a) BP

```
Active (vertex v){
    return metadata_curr[v]
    != metadata_prev[v];
}
Compute (edge e, weight w){
    old_dist = metadata_curr[e.dest];
    new_dist = metadata_curr[e.src] + w;
    return old_dist > new_dist ?
            new_dist: old_dist;
}
Combine (metadata_t *A){
    return min(A);
}
```

(b) SSSP

```
Active (vertex v){
    return v ∈ active layer;
}

Compute (edge e, weight w){
    if push   return metadata[src]*w
    else return ∂E/∂G_metadata · ∂metadata/∂metadata_prev · ∂metadata/∂w
}
Combine (metadata_t *A){
    if push  return sigmoid(sum(A));
    else      learning_rate *sum(A)
}
```

(c) MLP (E is the prediction errors of vertices in last layer)

Figure 3.5: ACC model with BP, SSSP, and MLP

differentiating the operations on a vertex, edge and update. At the meanwhile, task management is handled by SIMD-X, which will be presented in later sections.

3.4.2 Overview

The new ACC model contains three functions: **A**ctive, **C**ompute, and **C**ombine. ACC supports a wide range of graph algorithms and requires much fewer lines of code compared to prior work (Table 3.1). In this following, we will discuss the three functions.

Active allows a programmer to specify the condition whether a vertex is active. Formally it can be defined:

$$\exists_v \leftarrow active(M_v, v)$$

where v is the vertex ID and M_v represents its metadata. Depending on the algorithm, the Active function may vary. Figure 3.5 shows three examples. Belief propagation (BP) is simple which treats all vertices as active. In comparison, SSSP considers the vertices active when their current metadata differs from the prior iteration. On the other hand, multi-layer perceptron (MLP) takes a different view, where it views each neuron as one vertex and at each layer, all vertices belong to this layer become active.

To put it simply, SIMD-X distinguishes active vertices from inactive ones, and focuses on the calculation needed for each vertex. This is different from the vertex-centric model which deals with not only the active vertex but also its neighbors. Because two vertices may have different numbers of neighbors, existing systems [145, 112] likely suffer from workload imbalance. To this end, SIMD-X leverages a classification technique, similar to [1], to group the active vertices depending on the expected workload.

Compute defines the computation that happens on each edge. In particular, it specifies the operations on the metadata of edge $(v,\ u)$ and two vertices v and u, which can be written as follows:

$$update_{v \to u} \leftarrow compute(M_v, M_{(v,u)}, M_u)$$

where the return value of $update_{v \to u}$ will be used by the Combine function. For example, BP multiplies the edge weight with metadata and SSSP computes the updated distance for the destination vertex. For MLP, it multiplies the edge weight and metadata of input neuron in the push model, while computes the gradient descent [162] updates for edge weight in the pull model.

Combine merges all the updates, once the computations are completed. It can be represented:

$$update_u \leftarrow \bigoplus_{v \in Nbr[u]} update_{v \to u}$$

where $\oplus$ must be commutative and associative, e.g., sum and minimum, and is being applied to all the neighbors of vertex u. Figure 3.5 presents the Combine examples of BP, MLP, and SSSP. Particularly, BP summarizes all updates, where SSSP combines all updates from compute by selecting the minimum. Again, MLP experiences different cases in push and pull models. In push, it summarizes the value of all input neurons and conducts a sigmoid compute for the output. In pull, it summarizes the backward updates and multiplies with the learning rate for the weight update.

SIMD-X optimizes two types of combine operations, i.e., aggregation and voting. Particularly, aggregation cannot tolerate overwrites, that is, all updates are needed for computing the results. PageRank, SSSP and k-Core are representative examples of such operation. In contrast, voting relaxes this condition, that is, the algorithm is correct as long as one update is received because all updates are identical. BFS,

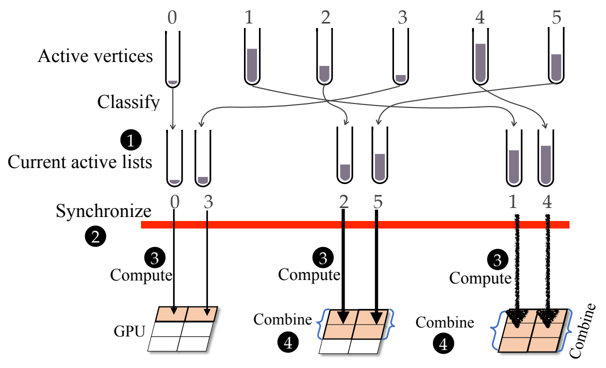


Figure 3.6: System-level view of the ACC model

weakly connected component and strongly connected component algorithms [62] fall into this category.

3.4.3 Work Flow

Here we present the work flow of SIMD-X, that is, how three ACC functions are executed on GPUs. On the high level, SIMD-X structures graph computation as a loop, each iteration of which can be viewed as a sequence of Figure 3.6. Specifically, SIMD-X utilizes the **Active** function ❶ to identify the active vertices and classify them into different groups based on anticipated workload, e.g., three groups of {0, 3}, {2, 7} and {1, 5} in this example. Here, synchronization ❷ is needed to assure that task generation is completed, which we will discuss in detail in Section 3.6. Next, SIMD-X maps each group of tasks to GPU cores for **Compute** ❹. Note, for all tasks in the same group, SIMD-X treats them like regular applications in Figure 3.2(a). For different groups, SIMD-X treats them differently, e.g., SIMD-X uses a **Combine** ❺ function for the tasks with large workloads that need to merge the results. Section 3.5 will further discuss how task management interacts with ❶ and ❹.

SIMD-X uses the pull-push model as in [71, 70, 1], by controlling where (in/out edge) Compute happens and how to apply (atomic or atomic free) the **Combine** result. Particularly, in the push model, SIMD-X conducts Compute on the out neighbors of each active vertex, and relies on atomic operations to apply the updates to the destination vertices. In contrast, the pull model schedules Compute on the in neighbors

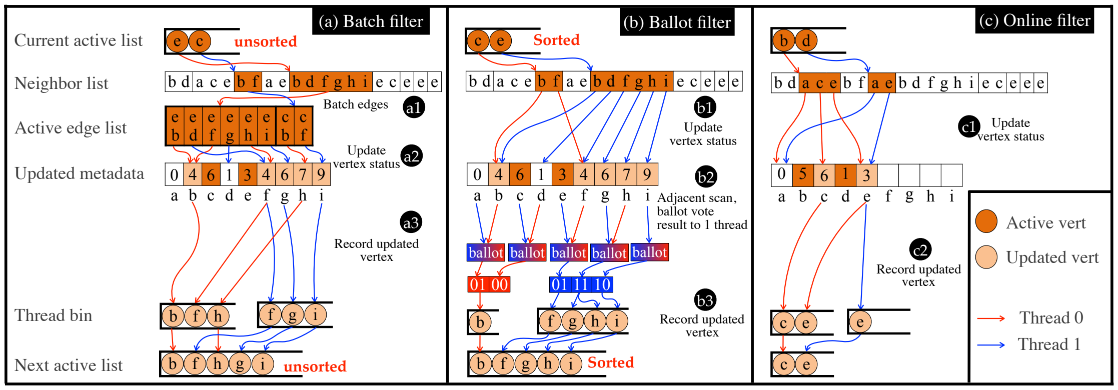


Figure 3.7: Three task management methods. Particularly, batch filter and ballot filter work on Figure 3.1(d) to produce a task list for next iteration. Online filter does that for Figure 3.1(c).

of active vertices, and uses atomic-free operations to update the destination vertices. As different iterations favor one model over the other, we follow a similar rule as in Ligra [71], that is, changing from the push to pull model when working on the push model works on more than 30% of the edges.

3.5 Just-In-Time Task Management

Task management is essential for graph applications. The key to success is to ensure good workload balance on GPUs, that is, each GPU core, regardless of from which streaming processor, accounts for a similar amount of the workload. To this end, SIMD-X utilizes just-in-time task management to filter inactive vertices and group comparable tasks together. In the following, we will first present the current approach of batch filter and analyze its drawbacks, and then describe two new filters, as well as the selection mechanism.

Batch Filter used in prior projects [75, 74] first loads all the edges of the active vertices to construct an active edge list, shown as step a1 in Figure 3.7. In the example of SSSP as shown in Figure 3.1(c) with two threads, the filter loads the neighbor lists of vertices $\{e, c\}$ to construct the active list of eight edges. Next, the filter checks these edges and updates vertex metadata a2, followed by recording the statuses of updated vertices a3 which becomes the task list for the next iteration, that is, $\{b,\ f,\ h,\ g,\ i\}$. A local storage per thread – thread bin – is used to avoid the expensive atomic operations, because multiple threads may concurrently put active vertices into the next active list. This way, the active vertices is stored in a thread bin first, and later the batch filter combines all thread bins to formulate the final active list.

We have observed several drawbacks when using the batch filter in various graph algorithms. First, the active list can consume up to 2·|E| memory space because the majority of the vertices in a graph can become active at one iteration [70, 1]. Considering GPU has very limited on-board memory (e.g., 16 GB), this restriction makes large-scale graph computing prohibitive. Second, batch filter produces an unsorted list of active vertices, which leads to poor memory performance. Lastly, this method may need to remove duplicated vertices in the active list because several threads from different source vertices may update the same destination vertex, e.g., vertex f in Figure 3.7(a).

Ballot Filter is designed to overcome all these shortcomings. It first loads the neighbors of active vertices and immediately updates vertex metadata b1. As shown in Figure 3.7(b), the neighbors of {e, c} get updated immediately. Comparing to the batch filter, this design not only drastically reduces the memory consumption, but also improves the memory performance through using coalesced scan and sorted active list.

In the batch filter, the vertex metatdata are updated after all the edges are gathered, in order to reduce TLB (translation lookaside buffer) pressure. Otherwise, the batch filter would access the neighbor list and metadata arrays simultaneously in a random fashion [74]. In contrast, the new ballot filter generates a sorted active list that leads to preferably sequential access of the neighbor list. This is done with *ballot* scan to compare the updated and previous metadata b2.

In the example in Figure 3.7, two threads perform coalesced scan of vertex metadata, and with the CUDA *__ballot*() primitive, return a bit variable '01' to the first thread. Here 1 means active and 0 otherwise, in this case, vertex a is inactive while b is. Through collaboratively working on the entire metadata array, the first thread

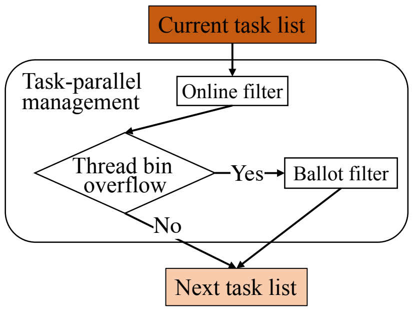


Figure 3.8: Just-in-time task management.

eventually gets the bit value ‘0100’ for the first four vertices, while the second thread ‘011110’ for the remaining six vertices. In this end, this approach produces a sorted active list, that is, $\{b, f, g, h, i\}$ b3.

Ballot filter is not without its own issue, especially in the case of a small number of active vertices. Here scanning the metadata array would account for the majority of the runtime. For instance, in ER and RC graphs, 99.23% and 96.67% of the time is spent on scanning metadata in ballot filter alone solution, respectively. As we will show shortly, it is desirable to improve the performance in this scenario.

Online Filter is designed in a way that it first loads the active neighbors, updates the destination vertex, and simultaneously records the active vertices in the thread bin c1. Afterwards, it assembles all thread bins together as the next active list c2. When the number of active vertices is small, this approach is much faster. Here we use an early stage of SSSP as an example, shown in Figure 3.7(c). In this example, $\{b, d\}$ are active vertices, and this approach loads their neighbors for computation, and immediately records the destination vertices. Eventually, it generates $\{e, c\}$ as the active list for the next iteration.

In graph computing, it is possible for a lot of vertices to become active at the same time, e.g., one BFS thread may work on more than 4,096 vertices for the Twitter graph. Clearly, one cannot afford such large a thread bin for each thread, thus online filter inevitably suffers from an overflow problem. The ballot filter largely avoids this issue because it first updates the metadata of active vertices b2, which, to some extent, averages out the active vertices across threads in step b3. Our evaluation

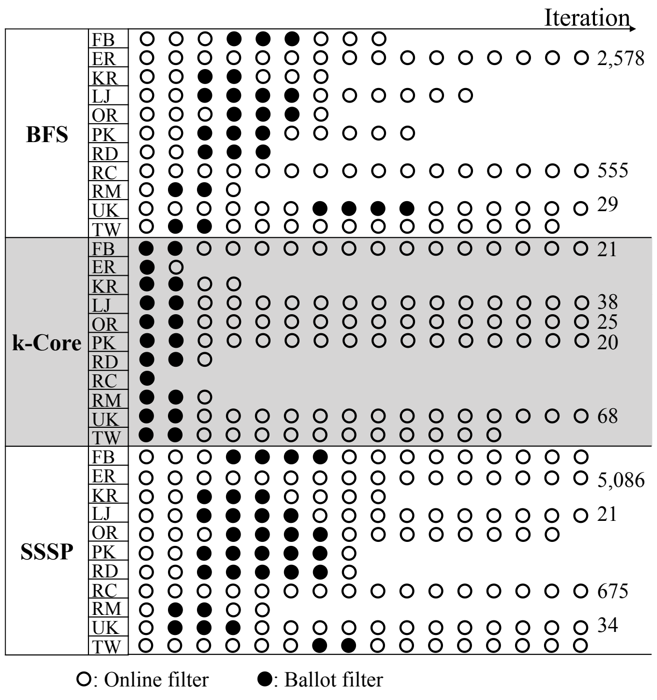


Figure 3.9: Ballot filter activation patterns.

also demonstrates such a difference in Figure 3.12. It is also important to note that for online filter, the vertices in the active list may become redundant, and can be out of order.

Just-In-Time Control is used in SIMD-X to determine the usage of ballot and online filter. As shown in Figure 3.8, SIMD-X always activates the online filter first. Once a thread bin overflows, SIMD-X will turn to the ballot filter to generate the correct task list for the next iteration. Interestingly, we find out that various algorithms and graph datasets present different selection patterns which tie closely to the amount of workload, that is, the higher volume of workload often results in the activation of ballot filter. As shown in Figure 3.9, BFS and SSSP typically use the ballot filter in the middle of the computation and online filter at the beginning and end. For high-diameter graphs, BFS and SSSP avoid the use of ballot filter. For instance, ER and RC always use the online filter along 2,578, 555, 5,086 and 675 iterations. k-Core activates the ballot filter at the initial iterations, i.e., typically the first two iterations except RC which only experiences one iteration because all its vertices have < 16 neighbors. At the extreme, MLP, BP, and PageRank need the ballot filter at

Kernel	Push (no fusion)				Pull (no fusion)				Selective fusion		All fusion
	Thread	Warp	CTA	Task mgt	Thread	Warp	CTA	Task mgt	push	pull	
Register consumption	26	27	28	24	24	24	22	30	48	50	110
# Kernel launch	up to 40,688								3		1

Table 3.2: Register consumption for various kernels.

exactly the first iteration of the computation.

3.6 Push-Pull Based Kernel Fusion

Kernel fusion is a common optimization for GPU applications [144, 1, 74, 75, 143, 161], which reduces expensive overhead of kernel invocation, as well as minimizes the global memory traffic as the life time of registers and shared memory is limited in each kernel.

When developing SIMD-X, we first start with an aggressive fusion approach, that is, creating one large kernel for the entire graph algorithm. However, this fused kernel requires a large number of the registers, and subsequently, supports fewer parallel warps which hurts the overall performance. As shown in Table 3.2, the register consumption (using the compilation flag *-Xptxas -v*) increases from average 25 to 110, that is 4.4× before and after kernel fusion.

It becomes clear that we need a balanced fusion strategy that keeps both register consumption and kernel invocation low. To this end, SIMD-X leverages the push-pull model used in the graph algorithms. That is, such algorithms often use push or pull based computing in several consecutive iterations. For example, BFS and SSSP utilize push in the first and last iterations, and pull in between. In contast, k-Core conducts pull at the beginning while push in the end.

The idea of push-pull based kernel fusion is to fuse kernels around the pull and push computing. In other words, for the push-based iterations, SIMD-X fuses different compute kernels (for thread, warp, CTA), as well as task management kernel, into one push kernel. The kernel only terminates when the computation finishes or it needs to switch to pull computing according to the criterion discussed in Section 3.4.3. Similar optimizations are done for the pull-based iterations.

Using the new push-pull based fusion, the register consumption decreases to 48

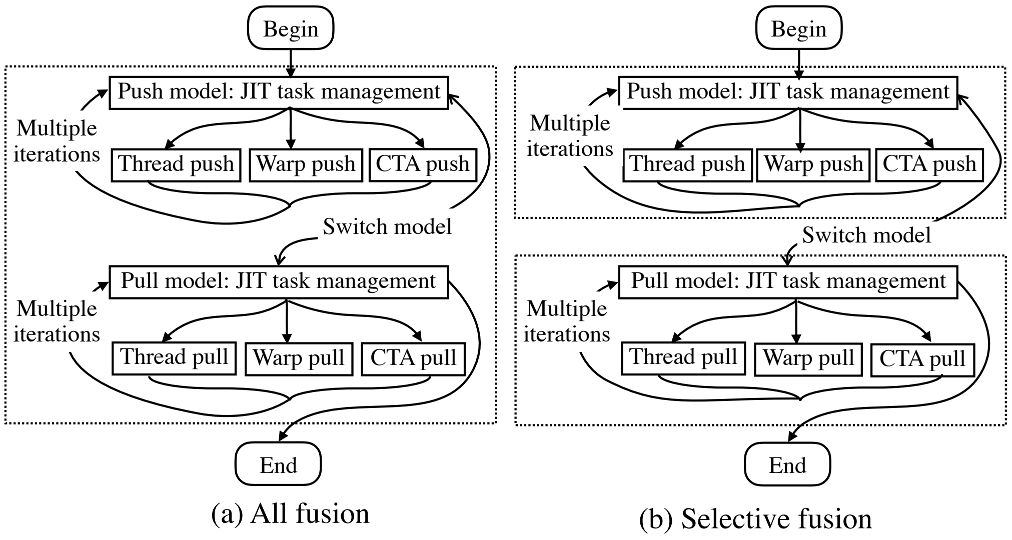


Figure 3.10: Graph computing typically clusters push and model computation separately together: (a) all fusion, (b) selective fusion.

and 55 thus increases the configurable thread count by 50%. Together, our evaluations demonstrate a 25% performance improvement.

Software global barrier is needed to enable the balanced kernel fusion. Generally speaking, this approach uses an array – *lock* – to synchronize all GPU threads upon arrival and departure. During the processing, it assumes the thread CTA as the monitor while the remaining threads as workers. At arrival, each worker CTA updates its own *status* in *lock*. Once all worker CTAs have arrived, the monitor changes the *statuses* of all CTAs to *departure*, allowing all threads to proceed forward.

This approach, unfortunately, suffers from potential deadlock [160], as illustrated in Figure 3.11. Specifically, the worker thread CTAs may hold all GPU hardware resources, such as streaming processors, registers and shared memory, while waiting for the monitor to update the *lock* array. In the meantime, the monitor cannot update the *lock* array, due to lack of hardware resources (e.g., thread over subscription).

Deadlock-free barrier. SIMD-X utilizes the barrier in a way to ensure that every CTA, regardless of a work or the monitor, can obtain hardware resources when needed. This is achieved through comparing the resources needed by the kernels, against the total available resources. Based on the GPU architecture, we can obtain the total amount of registers ($\#registerPerSMX$) that can be provided by each streaming

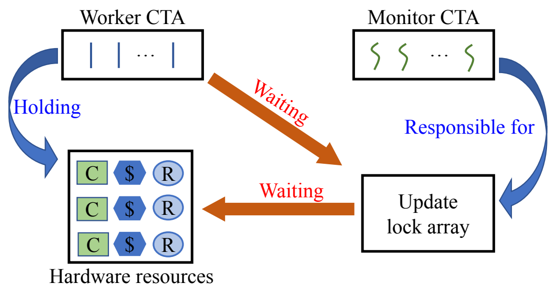


Figure 3.11: Deadlock problem in software global barrier, where 'C', '$', and 'R' represent CUDA core, L1 cache and register, respectively.

processor, e.g., 65,536 registers of NVIDIA K40 GPUs and 32,768 from K20 GPUs. On the other hand, we can collect the register consumption ($\#registerPerThread$) of each kernel at the compilation stage. Putting these numbers together, SIMD-X is able to calculate the appropriate thread configuration for the kernels.

The number of CTA can be computed as follows:

$$\#CTA = floor(\frac{\#registersPerSMX}{\#registersPerThread \cdot \#threadsPerCTA}) \cdot \#SMX \qquad (3.1)$$

where $\#threadsPerCTA$ is configured by a user, i.e., 128 by default. For example, when deploying a kernel, each thread consumes 110 registers, and on K40 that contains 15 SMXs, each of which contains 65,536 registers. If $\#threadsPerCTA$ is set to 128, one gets $\#CTA = ceil(\frac{65536}{110 \times 128}) \times 15 = 60$. As a result, we can configure this kernel as CTA and thread count per CTA as 60 and 128, respectively. On NVIDIA GPUs, CUDA provides a function *cudaOccupancyMaxPotentialBlockSize* which can be used to configure kernel threads, similar as equation (3.1).

Table 3.2 presents the register consumption and kernel invocation of different kernel fusion techniques. By using the push-pull based kernel fusion, the kernel relaunch is reduced to 3 while its register consumption is cut by half. In Figure 3.13, we will later show that this technique brings upto 80% performance benefit.

3.7 Graph Algorithms

This section presents a variety of algorithms which are implemented on SIMD-X to examine the expressiveness of ACC programming model, and performance impacts of the task management and kernel fusion techniques.

BFS [1] traverses a graph level by level. At each level, it loads all neighbors that are connected to vertices visited in the preceding level, inspects their statuses (metadata), and subsequently marks those unvisited neighbors as active for the next iteration. Notably, BFS conducts synchronizations at the end of each level, relies on vote to combine the updates. During the entire process of traversal, BFS typically experiences light workload at the beginning and end of the computation while heavy workload in the middle.

Belief propagation (BP), also known as sum-product message passing algorithm, infers the posterior probability of each event based on the likelihoods and prior probabilities of all related events. Once modeled as a graph (Bayesian network or Markov random fields), each event becomes a vertex with all incoming vertices and edges as related events and corresponding likelihoods. In BP, vertex possibility is the metadata.

k**-Core** (KC), which is widely used in graph visualization application [5, 48], iteratively deletes the vertices whose degree is less than k until all remaining vertices in this graph possess more than k neighbors. k-Core experiences large volume of workloads at initial iterations and follows with light workloads. This work uses a default value of $k = 16$.

Multi-layer perceptron (MLP) is a popular graphical machine learning algorithm that classifies a dataset into the corresponding categories, e.g., accounting for 61% of Tensorflow usage [163]. In this work, MLP considers the neurons, inter-neuron connections and connection weights as vertex, edge and weight sets, respectively.

PageRank (PR) [43] updates the rank value of one vertex based on the contribution of all in-neighbors iteratively till all vertices have stable rank values. Because the contributions of in neighbors are summarized to the destination vertex, we start

Graph Name	Abbrev.	Vertex Count	Edge Count
Facebook	FB	16,777,215	775,824,943
Europe-osm	ER	50,912,018	108,109,319
Kron24	KR	16,777,216	536,870,911
LiveJournal	LJ	4,847,571	136,950,781
Orkut	OR	3,072,626	234,370,165
Pokec	PK	1,632,803	61,245,127
Random	RD	4,000,000	511,999,999
RoadCA-net	RC	1,971,281	5,533,213
R-MAT	RM	3,999,983	511,999,999
UK-2002	UK	18,520,343	596,227,523
Twitter	TW	25,165,811	787,169,139

Table 3.3: Graph Dataset

PageRank with the pull model and *agg_sum* as the merge operation. At the end of PageRank, we switch to the push model because the majority of the vertices are stable [72]. The switch is decided by a decision tree.

Single source shortest path (SSSP) computes the shortest path between source vertex and the remaining vertices of the graph. Albeit similar to BFS as traversal algorithm, SSSP is more challenging mainly for the order of computing these active vertices is strictly restricted, that is, the vertex with the shortest distance should be computed first. To improve the parallelism, we adopt the delta-step [60] algorithm which allows us to compute the vertices whose distances are relatively shorter together.

Storage format. SIMD-X employs *compressed sparse row* (CSR) format to store the graph. For undirected graph, we only need to store the out-neighbors of each vertex. For directed graph, we store both out-neighbors and in-neighbors of each vertex to support the push and pull based processing.

3.8 Experiments

We have implement SIMD-X[1] with 5,660 lines of CUDA and C++ code. All the algorithms presented in Section 3.7 are implemented with around 100 lines of C++

[1]SIMD-X will be released in open source upon the paper publication.

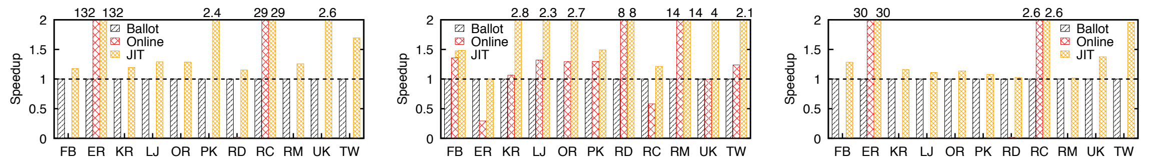
Alg	System	FB	EU	KR	LJ	OR	PK	RD	RC	RM	UK	TW
BFS	SIMD-X	165	400	130	38	37	39	56	15	57	80	249
	Gunrock	685 (4×)	849 (2×)	677 (5×)	71 (2.0×)	225 (6×)	44 (1.1×)	647 (12×)	146 (10×)	506 (9×)	112 (1.4×)	697 (2.8×)
	Galois	482 (2.9×)	1068 (2.7×)	140 (1.1×)	139 (3.7×)	42 (1.1×)	44 (1.1×)	48 (0.8×)	53 (3.5×)	65 (1.1×)	217 (2.7×)	322 (1.3×)
PR	SIMD-X	1553	346	350	236	435	118	1105	13	800	637	1525
	Gunrock	3004 (1.9×)	884 (2.6×)	3129 (8.9×)	275 (1.2×)	927 (2.1×)	166 (1.4×)	2963 (2.7×)	43 (3.3×)	2208 (2.8×)	784 (1.2×)	3180 (2.1×)
	Galois	4552 (2.9×)	603 (1.7×)	3069 (8.8×)	424 (1.8×)	1061 (2.1×)	218 (1.8×)	3576 (3.2×)	20 (1.5×)	2067 (2.6×)	842 (1.3×)	4178 (2.7×)
SSSP	SIMD-X	1703	1080	947	301	534	142	1669	223	485	678	1295
	Gunrock	OOM	1206 (1.1×)	1220 (1.3×)	431 (1.4×)	1259 (2.4×)	336 (2.4×)	5059 (3×)	229 (1.03×)	OOM	OOM	OOM
	Galois								66 (0.3×)			
KC	SIMD-X	366	77	130	59	63	32	230	14	19	151	277
	Gunrock								56 (4×)			
	Galois								66 (4.7×)			

Table 3.4: Runtime (ms) of SIMD-X and Gunrock and Galois. A K40 GPU is used to test SIMD-X and Gunrock, and a CPU with 28 threads for Galois. The blank space is the test cannot complete for the given algorithm and graph.

code. The source code is compiled by GCC 4.8.5 and NVIDIA nvcc 7.5 with the optimization flag as O3. In this work, we evaluate SIMD-X on a Linux workstation with two Intel Xeon E5-2683 CPUs (14 physical cores with 28 hyperthreads), and 512GB main memory. Throughout the evaluation, we use uint32 as the vertex ID and uint64 as index and evaluate our system on NVIDIA K40 GPUs unless otherwise is specified. We also test SIMD-X on earlier K20 and latest P100 GPUs. The timing is started once the graph data is loaded in GPU global memory. Each result is reported with an average of 64 runs.

3.8.1 Graph Benchmarks

We evaluate on a wide range of graphs as shown in Table 3.3, which falls into four types, i.e., social networks, road maps, hyperlink web and synthetic graphs. Particularly, Facebook [164], LiveJournal [137], Orkut [137], Pokec [137], and Twitter [7] are common social networks. Europe-osm [165] and RoadCA-net [136] are two large roadmap graphs, and UK-2002 [136] is a web graph. Furthermore, we use Graph500 generator to generate Kron24 [69], and GTgraph [138] for R-MAT and random graphs. Europe-osm and RoadCA-net are high diameter graphs, with 2570 and 555 as their



(a) BFS (b) k-Core (c) SSSP

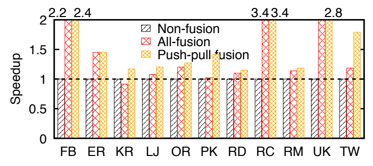
Figure 3.12: Benefit of just-in-time task management, normalized to the performance of the ballot filter.

diameters, respectively. LiveJournal, Pokec, Twitter and UK-2002 are medium diameter graphs, i.e., 10 - 30 as their diameters. The diameters of the remaining graphs are all smaller than 10. For graphs without edge weight, we use a random generator to generate one weight for each edge similar to Gunrock [75].

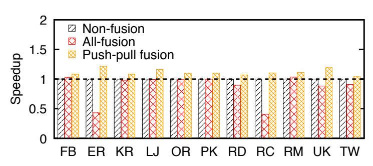
3.8.2 Comparison with State-of-the-art

Table 4.13 summarizes the runtime of SIMD-X against Galois and Gunrock which are state-of-the-art CPU and GPU graph processing systems. In general, SIMD-X outperforms Gunrock and Galois on K40 GPUs by 4× and 3.2×, respectively. In BFS, SIMD-X bests Gunrock and Galois by 5× and 2×. The best and worst speedups over Gunrock are achieved on the RD and PK graphs of 11.6× and 1.1×, while for Galois, the LJ and RD graphs of 3.7× and 0.85×. SIMD-X is slower than Galois on the RD graph, because task management brings no benefit to a uniform graph. On the other hand, Gunrock performs worse on this graph due to the larger overhead of the batch filter.

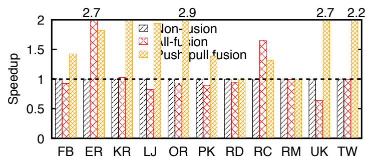
In PageRank, SIMD-X achieves over 2.5× speedups compared to both Gunrock and Galois. For SSSP and k-Core, we have observed Gunrock and Galois either run out of memory (OOM), or stuck in the execution. SIMD-X achieves 1.7× speedup over Gunrock in SSSP while Galois because of low cost of atomic operation runs much faster than K40 based SSSP but is slower than P100 GPUs which only consumes 50 ms as we will discuss shortly. For k-Core,SIMD-X is 4× and 4.7× faster than Gunrock and Galois, respectively on RoadCA-net graph.



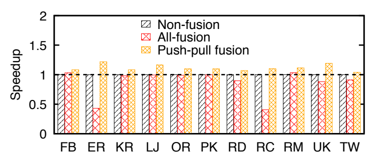
(a) BFS



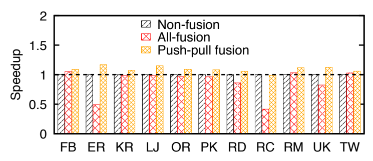
(b) BP



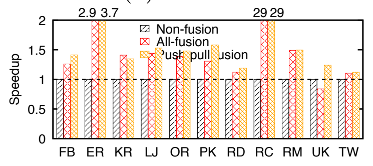
(c) k-Core



(d) MLP



(e) PageRank



(f) SSSP

Figure 3.13: Benefit of push-pull based kernel fusion, normalized to the performance of no fusion.

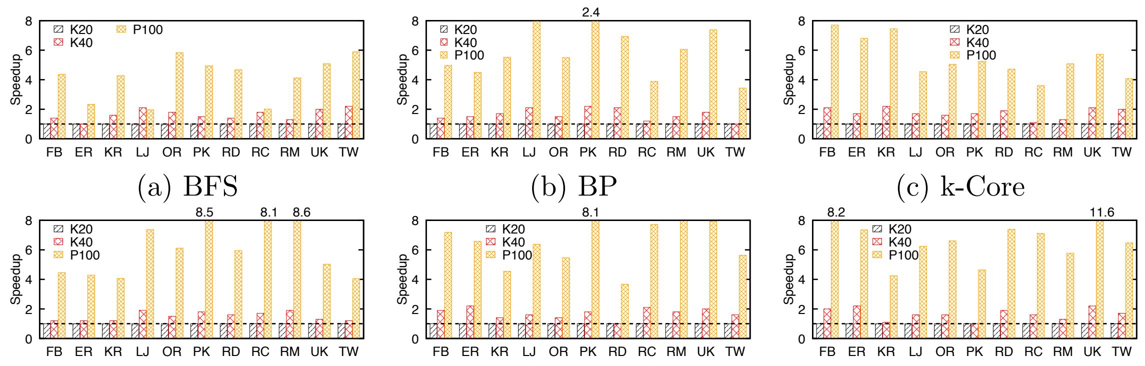
3.8.3 Benefits of Various Techniques

This section studies the performance impacts brought by JIT task management and push-pull based kernel fusion. As we have presented in Section 3.5, JIT task management only works for applications that experience workload variations, that is, BFS, k-Core and SSSP. On the other hand, push-pull based kernel fusion is applicable for all six algorithms

On average, JIT task management presented in Figure 3.12, is 16×, 26× and 4.5× faster than the ballot filter for BFS, k-Core and SSSP. As expected, online filter alone cannot work for many graphs, particularly large ones, e.g., Facebook, Twitter and UK2002 graphs in BFS and SSSP. Without considering overflow problem (ER and RC graphs), JIT task management adds a small 1-2% overhead on top of the online filter on BFS and SSSP.

On k-Core, JIT task management is, on average, 28.5× and 5% faster than ballot and online filter, respectively. We also observe that the ballot filter outperforms the online filter on ER and RC graphs by 3.4× and 1.7×, because k-Core removes a large volume of vertices which favors the former to produce a non-redundant and sorted active list.

Push-pull based kernel fusion brings average 43% and 25% improvement over non-fusion and all-fusion across all algorithms and graphs. In particular, push-pull based kernel fusion tops non-fusion by 74%, 11%, 85%, 8%, 10% and 66% on BFS, BP, k-



(a) BFS (b) BP (c) k-Core (d) MLP (e) PageRank (f) SSSP

Figure 3.14: SIMD-X performance differences on K20, K40 and P100 GPUs normalized to K20 performance.

Core, MLP, PageRank and SSSP. BFS, k-Core and SSSP achieves more performance gains because they are not computation intensive and tend to run a higher number of iterations. For all fusion, our new kernel fusion is 55%, 6%, 62%, 12%, 25% and 11% faster on BFS, BP, k-Core, MLP, PageRank and SSSP. It is important to note that all fusion is not always beneficial, i.e., all fuse option of MLP and PageRank is average 4% and 13% slower than no fusion, respectively. However, for high computation iterations, like BFS and SSSP on ER and RC, all fusion is on average 2× better.

3.8.4 Comparison of GPUs

We also evaluate SIMD-X on different types of GPUs, i.e., K20, K40 and P100 . Note that P100 [166] provides 3× computing power and 2× memory bandwidth than K40 GPUs. Interestingly, SIMD-X runs about 3.7× faster on P100 over K40. This indicates that our JIT task management and kernel fusion can leverage both hardware improvements on memory bandwidth and computing capability for faster graph computing. In comparison, Gunrock benefits only from bandwidth increase, achieving around 2.7× speedup from K40 to P100 [167].

Overall, six algorithms perform similarly on different GPUs. On average, K40 and P100 are 1.7× and 5.8× faster than K20 GPUs. Particularly, K40 is 1.6×, 1.6×, 1.8×, 1.7×, 1.5× and 1.7× better than K20 on BFS, BP, k-Core, PageRank, MLP and SSSP, respectively. For P100, it outperforms K20 by 4.1×, 6×, 5.5×, 6.5×, 6.1×, and 6.9× on BFS, BP, k-Core, PageRank, MLP and SSSP, respectively. It is important to note that SIMD-X on P100 GPUs is 18× and 17× better compared to the aforementioned results on Gunrock and Galois.

Chapter 4

Graphene: Fine-Grained IO Management for Graph Computing

4.1 Introduction

Graphs are powerful data structures that have been used broadly to represent the relationships among various entities (e.g., people, computers, and neurons). Analyzing massive graph data and extracting valuable information is of paramount value in social, biological, healthcare, information and cyber-physical systems [168, 169, 170, 171, 172, 173, 174].

Generally speaking, graph algorithms include reading the *graph data* that consists of a list of neighbors or edges, performing calculations on vertices and edges, and updating the *graph (algorithmic) metadata* that represents the states of vertices and/or edges during graph processing. For example, breadth-first search (BFS) needs to access the adjacency lists (data) of the vertices that have just been visited at the prior level, and mark the statuses (metadata) of previously unvisited neighbors as visited. Accesses of graph data and metadata come hand-in-hand in many algorithms, that is, reading one vertex or edge will be accompanied with access to the corresponding metadata. It is important to note that in this paper we use the term *metadata* to refer to the key data structures in graph computing (e.g., the statuses in BFS and the ranks in PageRank).

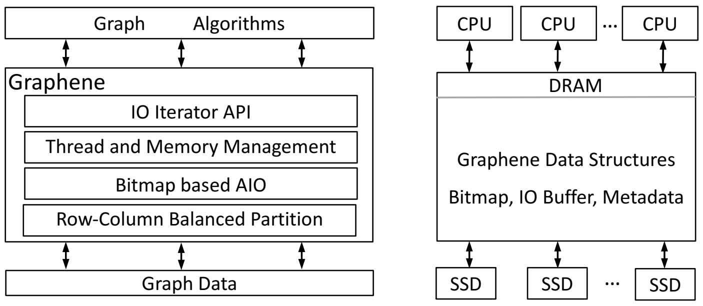


Figure 4.1: Architecture overview.

To tackle the IO challenge in graph analytics, prior research utilizes in-memory processing that stores the whole graph data and metadata in DRAM to shorten the latency of random accesses [146, 112, 6, 71, 151]. In-memory processing brings a number of benefits including easy programming and high-performance IOs. However, this approach is costly and difficult to scale, as big graphs continue to grow drastically in size. On the other hand, the alternative approach of external memory graph processing focuses on accelerating data access on storage devices. However, this approach suffers not only from complexity in programming and IO management but also slow IO and overall system performance [6, 76].

To close the gap between in-memory and external memory graph processing, we design and develop *Graphene*, a new semi-external memory processing system that efficiently reads the graph data on SSDs while managing the metadata in DRAM. Simply put, Graphene incorporates graph data awareness in IO management behind an IO centric programming model, and performs fine-grained IOs on flash-based storage devices. This is different from current practice of issuing large IOs and relying on operating system (OS) for optimization [6, 71, 76]. Figure 4.1 presents the system architecture. The main contributions of Graphene are four-fold:

IO (request) centric graph processing. Graphene advocates a new paradigm where each step of graph processing works on the data returned from an IO request. This approach is unique from four types of existing graph processing systems: (1) vertex-centric programming model, e.g., Pregel [145], GraphLab [146], PowerGraph [112], and Ligra [71]; (2) edge-centric, e.g., X-stream [151] and Chaos [153]; (3) embedding-centric, e.g., Arabesque [175]; and (4) domain-specific language, e.g., Galois [6], Green-Marl [152] and Trinity [150]. All these models are designed to address the complexity of the computation, including multi-threaded processing [6, 152], workload balancing [112, 176], inter-thread (node) communication [177] and synchronization [145]. However, in order to achieve good IO performance, these models require a user to explicitly manage the IOs, which is a challenging job by itself. For example, FlashGraph needs user input to sort, merge, submit and poll IO requests [76].

In Graphene, IO request centric processing (or IO centric for short) aims to simplify not only graph programming but also the task of IO management. To this end, we design a new *IoIterator* API that consists of a number of system and user-defined functions. As a result, various graph algorithms can be written in about 200 lines of code. Behind the scenes, Graphene translates high-level data accesses to fine-grained IO requests for better optimization. In short, IO centric processing is able to retain the benefit of easy programming while delivering high-performance IO.

Bitmap based, asynchronous IO. Prior research aims to read a large amount of graph data as quickly as possible, even when only a portion of it is needed. This design is justified because small random accesses in graph algorithms are not the strong suit of rotational hard drives. Notable examples include GraphChi [147] and X-stream [151], which read the entire graph data sequentially from the beginning to the end during each iteration of the graph calculation. In this case, the pursuit of high IO bandwidth overshadows the usefulness of data accesses. Besides this full IO model, the IO on-demand approach loads only the required data in memory, but again requires significant programming effort [149, 76, 178].

With the help of IO centric processing, Graphene pushes the envelope of the IO on-demand approach. Specifically, Graphene views graph data files as an array of 512-byte blocks, a finer granularity than more commonly used 4KB, and uses a *Bitmap*-based approach to quickly reorder, deduplicate, and merge the requests. While it incurs 3.4% overhead, the Bitmap approach improves the *IO utility* by as much as 50%, and as a result runs more than four times faster than a typical list based IO. In this work, IO utility is defined as the ratio between the amount of data

Type	Name	Return value	Description
System	Iterator->Next()	io_block_t	Get the next in-memory data block
	Iterator->HasMore()	bool	Check if there are more vertices available from IO
	Iterator->Current()	vertex	Get the next available vertex v
	Iterator->GetNeighbors(vertex v)	vertex array	Get the neighbors for the vertex v
User	IsActive(vertex v)	bool	Check if the vertex v is active
	Compute(vertex v)		Perform algorithm specific computation

Table 4.1: IoIterator API

that is loaded and useful for graph computation, and that of all the data loaded from disk. Furthermore, Graphene exploits *Asynchronous IO* (AIO) to submit as many IO requests as possible to saturate the IO bandwidth of flash devices.

Direct hugepage support. Instead of using 4KB memory pages, Graphene leverages the support of *Direct HugePage* (DHP), which preallocates the (2MB and 1GB) hugepages at boot time and uses them for both graph data and metadata structures, e.g., IO buffer and Bitmap. For example, Graphene designs a hugepage based memory buffer which enables multiple IO requests to share one hugepage. This technique eliminates the runtime uncertainty and high overhead in the transparent hugepage (THP) method [179], and significantly lowers the TLB miss ratio by 177×, leading to, on average, 12% performance improvement across different algorithms and graph datasets.

Balanced data and workload partition. Compared to existing 2D partitioning methods which divide vertices into equal ranges, Graphene introduces a row-column balanced 2D partitioning where each partition contains an equal number of edges. This ensures that each SSD holds a balanced data partition, especially in the cases of highly skewed degree distribution in real-world graphs. However, a balanced data partition does not guarantee that the workload from graph processing is balanced. In fact, the computation performed on each partition can vary drastically depending on the specific algorithm. To address this problem, Graphene utilizes dedicated IO and computing threads per SSD and applies a work stealing technique to mitigate the imbalance within the system.

We have implemented Graphene with different graph algorithms and evaluated its performance on a number of real world and synthetic graphs on up to 16 SSDs. Our experiments show that Graphene outperforms several external memory graph systems by 4.3 to 20×. Furthermore, Graphene is able to achieve similar performance to in-memory processing with the exception of BFS.

This chapter is organized as follows: Section 4.2 discusses the related work. Section 4.3 presents the IO centric programming model. Section 4.4 discusses bitmap-based, asynchronous IO and Section 4.5 presents data and workload balancing techniques, and Section 4.6 describes hugepage support. Section 4.7 describes a number of graph algorithms used in this work. Section 4.8 presents the experimental setup and results.

4.2 Related Work

Recent years have seen incredible advances in graph computation, to name a few, in-memory systems [6, 152, 71], distributed systems [150, 112, 176, 177, 180, 72], external-memory processing [147, 151, 153, 76, 181, 145, 146, 154, 149, 182, 148, 183], and accelerator-based systems [143, 4, 1]. In this section, we compare Graphene with existing projects from three aspects: programming, IO, and partitioning.

Programming. Prior projects, regardless of *Think like a vertex* [145, 176, 4, 147], *Think like an edge* [151, 153, 182], *Think like an embedding* [175], or *Think like a graph* [159], center around simplifying computation related programming efforts. In comparison, Graphene aims for ease of IO management with the new IO iterator API.

IO optimization is the main challenge for external memory graph engines, for which Graphene develops a set of fine-grained IO management techniques, including using 512-byte IO block and bitmap-based selective IO. Our approach achieves high efficiency compared to full IO [151, 153, 147, 145]. Compared to GridGraph [148] and FlashGraph [76], Graphene introduces a finer grained method that supports global range IO adjustment and reduces IO requests by 3×. Also, Graphene shows that asynchronous IOs, when carefully managed, are very beneficial for external memory systems. While hugepages are not new to graph systems [6, 76], Graphene addresses the issue of potentially low memory utilization by constructing IO buffers to share hugepages.

```
while true do
    foreach vertex v do
        if IsActive(v) then
            handle = IO_Submit(v);
            IO_Poll(handle);
            Compute(the neighbors of v);
        end
    end
    level++;
end
```
Algorithm 1: BFS with user-managed IO.

```
while true do
    block = IoIterator→Next();
    while block→HasMore() do
        vertex v = block→Current();
        if IsActive(v) then
            Compute(block→GetNeighbors(v));
        end
    end
    level++;
end
```
Algorithm 2: IoIterator-based BFS.

Partition optimization. A variety of existing projects [112, 184, 148, 76] rely on conventional 2D partitioning [185] to balance the workload. In contrast, Graphene advocates that it is the amount of edges, rather than vertices, in a partition that determines the workload. The new row-column balanced partition can help achieve up to 2.7× speedup on a number of graph algorithms.

4.3 IO Request Centric Graph Processing

Graphene allows the system to focus on the data, be it a vertex, edge or subgraph, returned from an IO request at a time. This new IO (request) centric processing aims to provide the illusion that all graph data resides in memory, and delivers high IO performance through applying various techniques behind the scenes which will be described in next three sections.

To this end, Graphene develops an *IoIterator* framework, where a user only needs to call a simple *Next()* function to retrieve the needed graph data for processing. This allows the programmers to focus on graph algorithms without worrying about the IO complexity in semi-external graph processing. At the same time, by taking care of graph IOs, the IoIterator framework allows Graphene to perform disk IOs more efficiently in the background and make them more cache friendly. It is worth noting that the IO centric model can be easily integrated with other graph processing paradigms including vertex or edge centric processing. For example, Graphene has a user-defined *Compute* function that works on vertices.

At a high level shown in Figure 4.2, we insert a new IoIterator layer between the algorithm and physical IO. In this architecture, the processing layer is responsible for the control flow, e.g., computing what vertices of the graph should be active, and

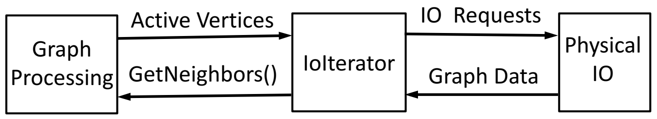


Figure 4.2: IoIterator programming model.

working on the neighbors of those active vertices. The IO layer is responsible for serving the IO requests from storage devices. Graph processing can start as soon as the IOs for the adjacency lists of the active vertices are complete, i.e., when the data for the neighbors become available. The new abstraction of IoIterator is responsible for translating the requests for the adjacency lists into the IO requests for data blocks.

Internally, Graphene applied a number of IO optimizations behind the IoIterator, including utilizing a Bitmap per device for sorting and merging, submitting large amounts of non-blocking requests via asynchronous IO, using hugepages to store graph data and metadata, and resolving the mismatch between IO and processing across devices.

The IoIterator layer consists of a set of APIs listed in Table 4.1. There are four system-defined functions for the IoIterator, *Next*, *HasMore*, *Current*, and *GetNeighbors*, which work on the list of the vertices returned from the underlying IO layer. In addition, two functions *IsActive* and *Compute* should be defined by the users. For example, in BFS, the IsActive function should return *true* for any frontier if a vertex v has been visited in the preceding iteration, and Compute should check the status of each neighbor of v, and mark any unvisited neighbors as frontiers for the next iteration. Detailed description of BFS and other algorithms can be found in Section 4.7.

An example of BFS pseudocode written with the current approach of user-managed selective IO vs. the IoIterator API can be found in Algorithms 1 and 2. In the first approach, the users are required to be familiar with the Linux IO stack and explicitly manage the IO requests such as IO submission, polling, and exception handling. The main advantage of the IoIterator is that it completely removes such a need. On the other hand, in both approaches, the users need to provide two similar functions, *IsActive* and *Compute*.

It is important to note that the pseudocode will largely stay the same for other algorithms, but with different *IsActive* and *Compute*. For example, in PageRank, IsActive returns *true* for vertices that have delta updates, and Compute accumulates the updates from different source vertices to the same destination vertex. Here, Compute may be written in vertex or edge centric model.

4.4 Bitmap Based, Asynchronous IO

Graphene achieves high-performance IO for graph processing through a combination of techniques including fine-grained IO blocks, bitmap, and asynchronous IO. Specifically, Graphene favors small, 512-byte IO blocks to minimize the alignment cost and improve the IO utility, and utilizes a fast bitmap-based method to reorder and produce larger IO requests, which will be submitted to devices asynchronously. As a result, the performance of graph processing improves as a higher fraction of useful data are delivered to CPUs at high speed.

In Graphene, graph data are stored on SSDs in *Compressed Sparse Row* (CSR) format which consists of two data structures: the *adjacency list array* that stores the IDs of the destination vertices of all the edges ordered by the IDs of the source vertices, and the *beginning position array* that maintains the index of the first edge for each vertex.

4.4.1 Block Size

One trend in modern operating systems is to issue IOs in larger sizes, e.g., 4KB by default in some Linux distributions [186, 187, 188, 189, 190, 191, 192]. While this approach is used to achieve high sequential bandwidth from underlying storage devices like hard drives, doing so as in prior work [76] would lead to low IO utility because graph algorithms inherently issue small data requests. In this work, we have studied the IO request size when running graph algorithms on Twitter [11] and Friendster [193]. Various graph datasets that are used in this paper is summarized in Section 4.8. One can see that most (99%) of IO requests are much smaller than 4KB

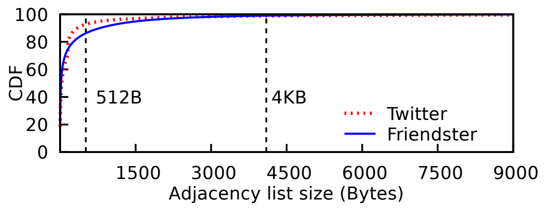


Figure 4.3: Distribution of IO sizes.

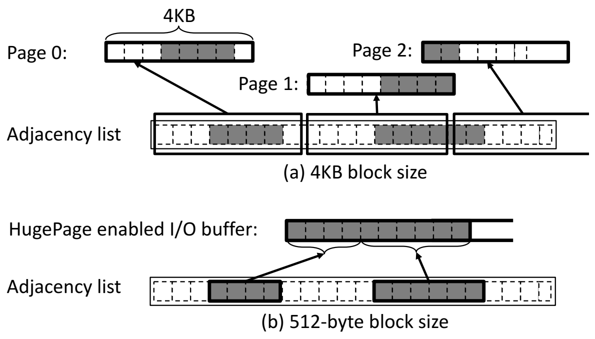


Figure 4.4: IO alignment cost: 4KB vs. 512-byte blocks, where one dotted box represents one 512-byte block.

as shown in Figure 4.3. Thus, issuing 4KB IOs would waste a significant amount of IO bandwidth.

In Graphene, we choose to use a small IO size of 512 bytes as the basic block for graph data IOs. Fortunately, new SSDs are capable of delivering good IOPS for 512-byte read requests for both random and sequential IOs. For example, Samsung 850 SSD [194], which we use in the experiments, can achieve more than 20,000 IOPS for 512-byte random read.

Another benefit of using 512-byte blocks is to lower the cost of the alignment for multiple requests. Larger block size like 4KB means the offset and size of each IO request should be a multiple of 4KB. In the example presented in Figure 4.4, requesting the same amount of data will lead to the different numbers of IOs when using 4KB (top) and 512-byte (bottom) block sizes. One can see that the former will load 2.2× more data, i.e., 12KB vs. 5KB in this case. In addition, combined with hugepage support that will be presented shortly, 512-byte block IO will need only one hugepage-based IO buffer, compared to three 4KB pages required in the top case.

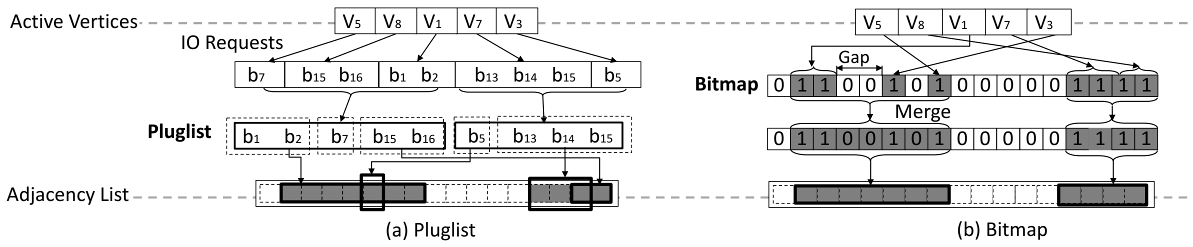


Figure 4.5: Pluglist vs. bitmap IO management, (a) Pluglist where sorting and merging are limited to IO requests in the pluglist. (b) Bitmap where sorting and merging are applied to all IO requests.

4.4.2 Bitmap-Based IO Management

At each iteration of graph processing, graph algorithms compute and generate the requests for the adjacency lists (i.e., the neighboring vertices) of *all* active vertices for the following iteration. In particular, Graphene translates such requests into a number of 512-byte aligned IO blocks, which are quickly identified in a new *Bitmap* data structure. In other words, Graphene maintains a Bitmap per SSD, one bit for each 512-byte block on the disk. For each request, Graphene marks the bits for the corresponding blocks, that is, should a block need to be loaded, its bit is marked as "1", and "0" otherwise. Clearly, the Bitmap offers a *global* view of IO operations and enables optimization opportunities which would not otherwise be possible.

For a 500GB SSD as we have used in this work, the size of the bitmap is merely around 128MB, which we can easily cache in CPUs and store in DRAM with a number of hugegages. Because Graphene combines Bitmap-based management with asynchronous IO, it is also able to utilize one IO thread per SSD. Therefore, since there is only one thread managing the Bitmap for each SSD, no lock is required on the Bitmap structures.

Issues with local IO optimization. Traditionally, the OS takes a *local* view of the IO requests by immediately issuing the requests for the neighbors of one or a group of active vertices. In addition, the OS performs several important tasks such as IO batching, reordering and merging at the block layer. Unfortunately, these techniques have been applied only to IO requests that have been buffered in certain data structures. For instance, Linux exploits a linked list called *pluglist* to batch and submit the IO requests [186], in particular, the most recent Linux kernel 4.4.0 supports 16 requests in a batch.

Figure 4.5(a) presents the limitations of the pluglist based approach. In this example, vertices $\{v_5, v_8, v_1, v_7, v_3\}$ are all active and the algorithm needs to load their neighbors from the adjacency list file. With a fixed-size pluglist, some of the requests will be *batched* and enqueued first, e.g., the requests for the first three vertices $\{v_5, v_8, v_1\}$. In the second step, *sorting* is applied across the IO requests in the pluglist. Since the requests are already grouped, sorting happens within the boundary of each group. In this case, the requests for the first three vertices are reordered from $\{b_7, b_{15}, b_{16}, b_1, b_2\}$ to $\{b_1, b_2, b_7, b_{15}, b_{16}\}$. In the third step, if some IO blocks present good spatial locality, *merging* will be applied to form a larger IO request, e.g., blocks $\{b_1, b_2, b_7\}$ are merged into one IO transaction. And later, a similar process happens for the IOs on the rest of vertices $\{v_7, v_3\}$.

In this case, there are four independent IO requests to the disk, (a) blocks b_1 - b_7, (b) blocks b_{15} - b_{16}, (c) block b_5, and (d) blocks b_{13} - b_{15}. The first request loads seven sequential blocks in one batch, which takes advantage of prefetching and caching and is preferred by the disks and OS. As a result, the third request for block b_5 will likely hit in the cache. On the other hand, although the second and fourth requests have overlapping blocks, they will be handled as two separate IO requests.

Bitmap and global IO optimization. Graphene chooses to carry out IO management optimizations, including IO deduplication, sorting and merging, on a *global* scale. This is motivated by the observation that although graph algorithms tend to present little or no locality in a short time period, there still exists a good amount of locality within the entire processing window. Bitmap-based IO management is shown in Figure 4.5(b). Upon receiving the requests for all active vertices, Graphene will convert the needed adjacency lists into the block addresses and mark those blocks in the Bitmap.

Sorting. The process of marking active blocks in the corresponding locations in the Bitmap naturally sorts the requests in the order of physical addresses on disks. In other words, the order of the requests is simply that of the marked bits in the Bitmap.

IO deduplication is also easily achieved in the process. Bitmap-based IO ensures that only one IO request will be sent even when the data block is requested multiple times, achieving the effect of IO deduplication. This is common in graph computation. For example, in the single source shortest path algorithm, one vertex may have many neighboring vertices, and if more than one neighbors need to update the distance of this vertex, it will need to be enqueued multiple times for the next iteration. In addition, different parts of the same IO block may need to be loaded at the same time. In the prior example, as the block b_{15} is shared by the requests from vertices v_7 and v_8, it will be marked and loaded once. Our study shows that the deduplication enabled from Bitmap can save up to 3× IO requests for BFS, compared to a pluglist based method.

IO merging. Bitmap is very easy to use for merging the requests in the vicinity of each other into a larger request, which reduces the total number of IO requests submitted to disks. For example, as shown in Figure 4.5(b), IO requests for vertices v_1, v_3, v_5 (and similarly for vertices v_7 and v_8) are merged into one. As a result, there are only two non-overlapping requests instead of four as in the pluglist case.

How to merge IO requests is guided by a number of rules. It is straightforward that consecutive requests should be merged. When there are multiple non-consecutive requests, we can merge them when the blocks to be loaded are within a pre-defined *maximum gap*, which determines the largest distance between two requests. Note that this rule directly evaluates the Bitmap by bytes to determine whether eight consecutive blocks are needed to be merged.

This approach favors larger IO sizes and has proven to be effective in achieving high IO performance. Figure 4.6 shows the performance when running BFS on the Twitter and UK graphs. Interestingly, the performance peaks for both graphs when the maximum gap is set to 16 blocks (i.e., 8KB). Graphene also imposes an upper bound for IO size, so that the benefit of IO merging would not be dwarfed by handling of large IO requests. We will discuss this upper bound shortly.

In conclusion, Bitmap provides a very efficient method to manage IO requests for graph processing. We will show later that while the OS already provides similar

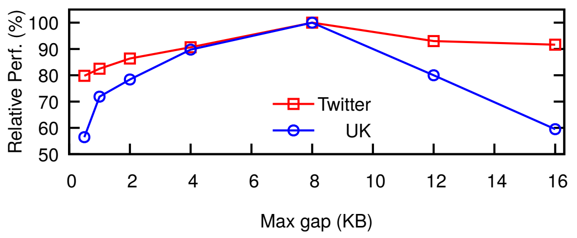


Figure 4.6: Graphene BFS Performance of maximum gap.

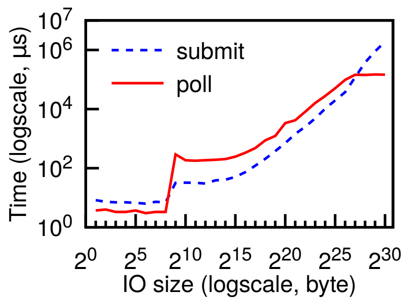
functionality, this approach is more beneficial for dealing with random IOs to a large amount of data. Besides Bitmap-based IO, we have also implemented a Pluglist based approach that extends the pluglist to support sorting, deduplication and merging in a global scale. As shown in Section 4.8, compared to a list, the Bitmap approach incurs smaller overhead and runs four times faster. It is important to note that although we focus on using Bitmap for graph processing in this work, it can also be applied to other applications. We will demonstrate this potential in Section 4.8.

4.4.3 Asynchronous IO

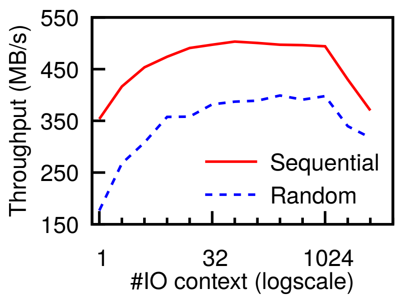
Asynchronous IO (AIO) is often used to enable a user-mode thread to read or write a file, while simultaneously carrying out the computation [186]. The initial design goal is to overlap the computation with non-blocking IO calls. However, because graph processing is IO bound, Graphene exploits AIO for a different goal of submitting as many IO requests as possible to saturate the IO bandwidth of flash devices.

There are two popular AIO implementations, i.e., user-level *POSIX AIO* and kernel-level *Linux AIO*. We prefer the latter in this work, because POSIX AIO forks child threads to submit and wait for the IO completion, which in turn has scalability issues while submitting too many IO requests [186]. In addition, Graphene leverages direct IO to avoid the OS-level page cache during AIO, and the possible blocks introduced by the kernel [195].

Upper bound for IO request. Although disks favor large IO sizes in tens or hundreds of MBs, it is not always advantageous to do so, especially for AIO. Typically, an AIO consists of two steps, submitting the IO request to an IO context and polling the context for completion. If IO request sizes are too big, the time for IO submission



(a) IO size



(b) IO context

Figure 4.7: AIO performance w.r.t. IO size and IO context

would take longer than polling, at which point AIO would essentially become blocking IO. Figure 4.7(a) studies the AIO submission and polling time. As the size goes beyond 1MB, submission time increases quickly. And once it reaches 128MB, it becomes blocked IO as submission time eventually becomes longer then polling time. In this work, we find that a modest IO size, such as 8, 16, and 32 KB, is able to deliver good performance for various graph algorithms. Therefore, we set the default upper bound of IO merging as 16KB.

IO context. In AIO, each IO context loads the IO requests sequentially. Graphene uses multiple contexts to handle the concurrent requests and overlap the IO with the computation. For example, while a thread is working on the request returned from one IO context, another IO context can be used to serve other requests from the same SSD. Given its intensive IO demand, graph computation would normally need to create a large number of IO contexts. However, without any constraints, too many IO contexts would hurt the performance because every context needs to register in the kernel and may lead to excessive overhead from polling and management.

Figure 4.7(b) evaluates the disk throughput with respect to the number of total IO contexts. As one can see that each SSD could achieve the peak performance with 16 contexts but the performance drops once the total IO context goes beyond 1,024 contexts. In this work, depending on the number of available SSDs, we utilize different numbers of IO contexts, by default using 512 contexts for 16 SSDs.

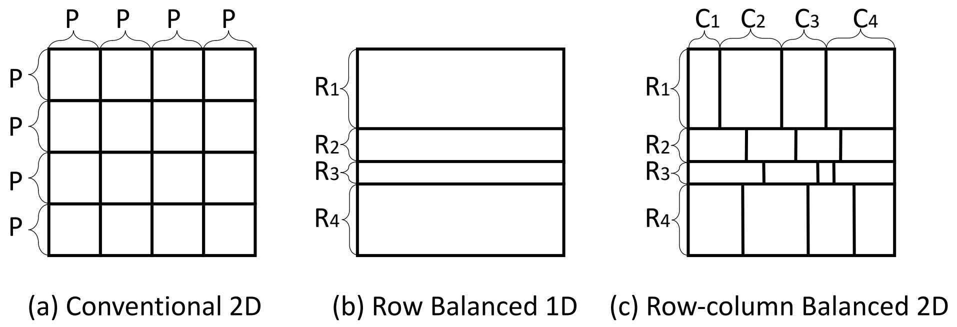


Figure 4.8: Graphene Balanced 2D partition.

4.4.4 Conclusion

In summary, combining 512-byte block and Bitmap-based IO management allows Graphene to load a smaller amount of data from SSDs, about 21% less than the traditional approach. Together with AIO, Graphene is able to achieve high IO throughput of upto 5GB/s for different algorithms on an array of SSDs.

4.5 Balancing Data and Workload

Taking care of graph data IO only solves half of the problem. In this section, we present data partitioning and workload balancing in Graphene.

4.5.1 Row-Column Balanced 2D Partition

Given highly skewed degree distribution in power-law graphs, existing graph systems, such as GridGraph [148], TurboGraph [149], FlashGraph [76], and PowerGraph [112], typically apply a simple 2D partitioning method [185] to split the neighbors of each vertex across multiple partitions. The method is presented in Figure 4.8(a), where each partition accounts for an equal range of vertices, P number of vertices in this case, on both row and column-wise. This approach needs to scan the graph data once to generate the partitions. The main drawback of this approach is that an equal range of vertices in each data partition do not necessarily lead to an equal amount of edges, which can result in workload imbalance for many systems.

To this end, Graphene introduces a row-column balanced 2D partitioning method,

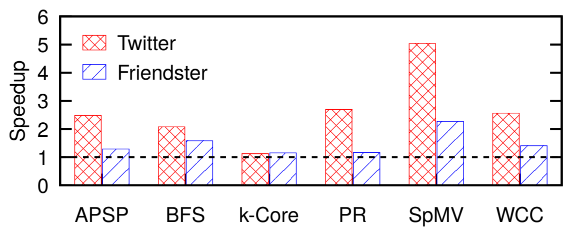


Figure 4.9: Benefit of row-column balanced 2D partition.

as shown in Figure 4.8(b-c), which ensures each partition contains an equal number of edges. In this case, each partition may have different numbers of rows and columns. This is achieved through three steps: (1) the graph is divided by the row major into R number of partitions, each of which has the same numbers of edges with potentially different number of rows; (2) Each row-wise partition is further divided by the column major into C number of (smaller) partitions, each of which again has the equal amount of edges. As a result, each partition may contain different number of rows and columns. Although it needs to read the graph one more time, it produces "perfect" partitions with the equal amount of graph data, which can be easily distributed to a number of SSDs.

Figure 4.9 presents the benefits of row-column balanced 2D partition for two social graphs, Twitter and Friendster. On average, the improvements are 2.7× and 50% on Twitter and Friendster, respectively. The maximum and minimum benefits for Twitter are achieved on SpMV for 5× and k-Core 12%. The speedups are similar for Friendster. While each SSD holds a balanced data partition, the workload from graph processing is not guaranteed to be balanced. Rather, the computation performed on each partition can vary drastically depending on the specific algorithm. In the following, we present the workflow of Graphene and how it balances the IO and processing.

4.5.2 Balancing IO and Processing

Although AIO, to some extent, enables the overlapping between IO and computation, we have observed that a single thread doing both tasks would fail to fully saturate the bandwidth of an SSD. To address this problem, one can assign multiple threads

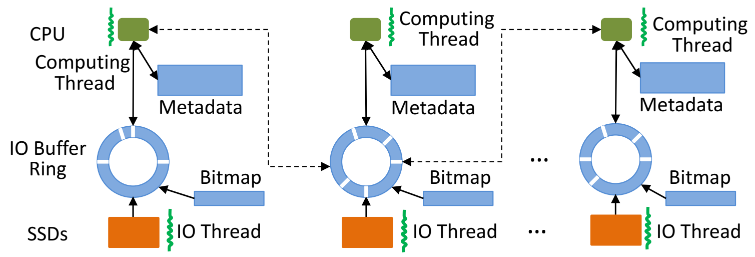


Figure 4.10: Graphene scheduling management.

to work on a single SSD in parallel. However, if each thread would need to juggle IO and processing, this can lead to contention in the block layer, resulting in a lower performance.

In Graphene, we assign two threads to collaboratively handle the IO and computation on each SSD. Figure 4.10 presents an overview of the workflow. Initially upon receiving updates to the Bitmap, a dedicated *IO thread* formulates and submits IO requests to the SSD. Once the data is loaded in memory, the *computing thread* retrieves the data from the *IO buffer* and works on the corresponding metadata. Using PageRank as an example, for currently active vertices, the IO thread would load their in-neighbors (i.e., the vertices with a directed edge to active vertices) in the IO buffer, further store them in the ring buffer. Subsequently, the computing thread uses the rank values of those in-neighbors to update the ranks of active vertices. The metadata of interest here is the rank array.

Graphene pins IO and computing threads to the CPU socket that is close to the SSD they are working on. This NUMA-aware arrangement reduces the communication overhead between IO thread and SSD, as well as IO and computing threads. Our test shows that this can improve the performance by 5% for various graphs.

Graphene utilizes a work stealing technique to mitigate computational imbalance issue. As shown in Figure 4.10, each computing thread first works on the data in its own IO buffer ring. Once it finishes processing its own data, this thread will check the IO buffer of other computing threads. As long as other computing threads have unprocessed data in IO buffers, this thread is allowed to help process them. This procedure repeats until all data have been consumed.

Figure 4.11 presents the performance benefit from work stealing. On average,

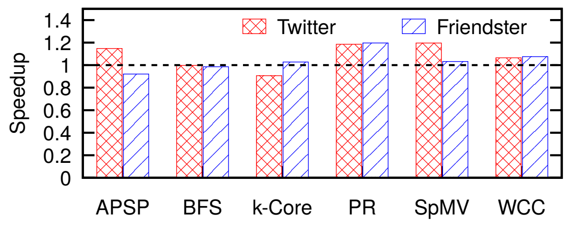


Figure 4.11: Benefit of workload stealing.

PageRank, SpMV, WCC and APSP achieve various speedup of 20%, 11%, 8% and 4%, respectively, compared to the baseline of not using workload stealing. On the other hand, BFS and k-Core suffer slowdown of 1% and 3%. This is mostly because the first four applications are more computation intensive while BFS and k-Core are not. One drawback of workload stealing is lock contention at the IO buffer ring, which can potentially lead to performance degradation, e.g., 8% for APSP on Friendster and k-Core on Twitter.

4.6 HugePage Support

Graphene leverages the support of *Direct HugePages* (DHP), which preallocates hugepages at boot time, to store and manage graph data and metadata structures, e.g., IO buffer and Bitmap, shown as blue boxes in Figure 4.10. This is motivated by our observation of high TLB misses, as the number of memory pages continues to grow for large-scale graph processing. Because a TLB miss typically requires hundreds of CPU cycles for the OS to go through the page table to figure out the physical address of the page, this would greatly lower the graph algorithm performance.

In Graphene, the OS creates and maintains a pool of hugepages at machine boot time when memory fragmentation is at the minimum. This is because any memory fragmentation would break physical space into pieces and disrupt the allocation of hugepages. We choose this approach over transparent hugepage (THP) in Linux [179] for a couple of reasons. First, we find that THP introduces undesirable uncertainty at runtime, because such a hugepage could be swapped out from memory [196]. Second, THP does not always guarantee successful allocation and may incur high CPU overhead. For example, when there were a shortage, the OS would need to aggressively compress the memory in order to provide more hugepages [197].

Data IO. Clearly, if each IO request were to consume one hugepage, a large portion of memory space would be wasted, because Graphene, even with IO merging, rarely issues large (2MB) IO requests. Alternatively, Graphene allows multiple IO requests to share hugepages. This consolidation is done through IO buffers in the IO Ring Buffer. Given a batch of IO requests, Graphene first claims a buffer that contains a varied number of continuous 2MB hugepages. As the IO thread works exclusively with a buffer, all IO requests can in turn use any portion of it to store the data. Also, consecutive IO requests will use continuous memory space in the IO buffer so that there is no fragmentation. Note that the system needs to record the begin position and length of each request within the memory buffer, which is later parsed and shared with the user-defined Compute function in the IoIterator. In addition, *direct IO* is utilized for loading disk blocks directly into hugepages. Comparing to buffered IO, this method skips the step of copying data to system pagecache and further to user buffer, i.e., double copy.

Metadata has been the focus of several prior works [184, 198, 185] to improve the cache performance of various graph algorithms. As a first attempt, we have investigated the use of page coloring [199, 200] to resolve cache contention, that is, to avoid multiple vertices being mapped to the same cache line. With 4KB pages, we are able to achieve around 5% improvement across various graphs. However, this approach becomes incompatible when we use 2MB hugepages for metadata, as the number of colors is determined by the LLC size (15MB), associativity (20) and page size.

To address this challenge, we decide to use hugepages for the metadata whose size is at the order of $O(|V|)$. In this work, we use 1GB hugepages, e.g., for PageRank, a graph with one billion vertices will need 4GB memory for metadata, that is, four 1GB hugepages.

This approach brings several benefits. Figure 4.12 illustrates the reduction in TLB miss introduced by this technique when running on a Kronecker graph. Across six algorithms, we observe an average 177× improvement with the maximum of 309× for PageRank. In addition, as prefetching is constrained by the page size, hugepages also

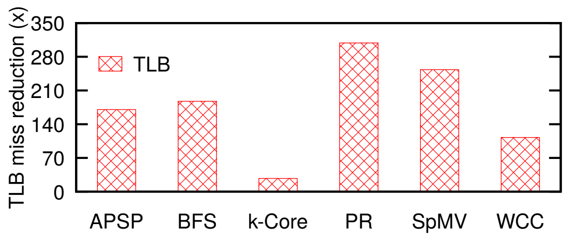


Figure 4.12: TLB misses reduced by hugepage-enabled buffer.

enables more aggressive hardware prefetching in LLC, now that the pages are orders of magnitude bigger (1GB vs. 4KB). The test shows that this technique provides around 10% speedup for these graph algorithms.

4.7 Graph Algorithms

Graphene implements a variety of graph algorithms to understand different graph data and metadata, and their IO patterns. For all the algorithms, the sizes of data and metadata are $O(|E|)$ (total count of edges) and $O(|V|)$ (total count of vertices), respectively.

Breadth First Search (BFS) [70, 1] performs random reads of the graph data, determined by the set of most recently visited vertices in the preceding level. The statuses (visited or unvisited) of the vertices are maintained in the status array, a key metadata in BFS. It is worthy to note that status array may experience more random IOs, because the neighbors for a vertex tend to have different IDs, some of which are far apart.

PageRank (PR) [43, 201] can calculate the popularity of a vertex by either pulling the updates from its in neighbors or pushing its rank to out neighbors. The former performs random IO on the rank array (metadata), whereas the latter requires sequential IO for graph data but needs locks while updating the metadata. In this work, we adapt delta-step PageRank [72], where only vertices with updated ranks should push their delta values to the neighbors, yet again requiring random IOs.

Weakly Connected Component (WCC) is a special type of subgraph whose vertices are connected to each other. For directed graphs, a strongly connected component exists if a directed path can be found between all pairs of vertices in the subgraph [61]. In contrast, a WCC exists if such a path can be found regardless of the edge direction. We implement the hybrid WCC detection algorithm presented in [62], that is, it uses BFS to detect the largest WCC then uses label propagation to compute remaining smaller WCCs. In this algorithm, the label array serves as the metadata.

k-Core (KC) [48, 202] is another type of subgraph where each vertex has the degree of at least k. Iteratively, a k-Core subgraph is found by removing the vertices from the graph whose degree is less than k. As the vertices are removed, their neighbors are affected, where the metadata – degree array – will need to be updated. Similar to aforementioned algorithms, since the degree array is indexed by the vertex IDs, the metadata IO in k-Core also tends to be random. k-Core is chosen in this work as it presents alternating graph data IO patterns across different iterations. Specifically, in the initial iterations, lots of vertices would be affected when a vertex is removed, thus the graph data is retrieved likely in the sequential order. However at the later iterations, fewer vertices will be affected, resulting in random graph data access.

All Pairs Shortest Path (APSP) calculates the shortest paths from all the vertices in the graph. With APSP, one can further compute Closeness Centrality and Reachability problems. Graphene combines multi-source traversals together, to reduce the total number of IOs needed during processing and the randomness exposed during the metadata access [49, 2]. Similar to FlashGraph, we randomly select 32 source vertices for evaluation to reduce APSP execution time on large graphs.

Sparse Matrix Vector (SpMV) multiplication exhibits sequential access when loading the matrix data, and random access for the vector. In this algorithm, the matrix and vector serve the role as graph data and metadata, respectively. As a comparison to BFS, SpMV is more IO friendly but equally challenging on cache efficiency.

Name	# Vertices	# Edges	Size	Preprocess (seconds)
Clueweb	978M	42.6B	336GB	334
EU	1071M	92B	683GB	691
Friendster	68M	2.6B	20GB	3
Gsh	988M	33.8B	252GB	146
Twitter	53M	2.0B	15GB	2
UK	788M	48B	270GB	240
Kron30	1B	32B	256GB	141
Kron31	2B	1T	8TB	916

Table 4.2: Graph Datasets.

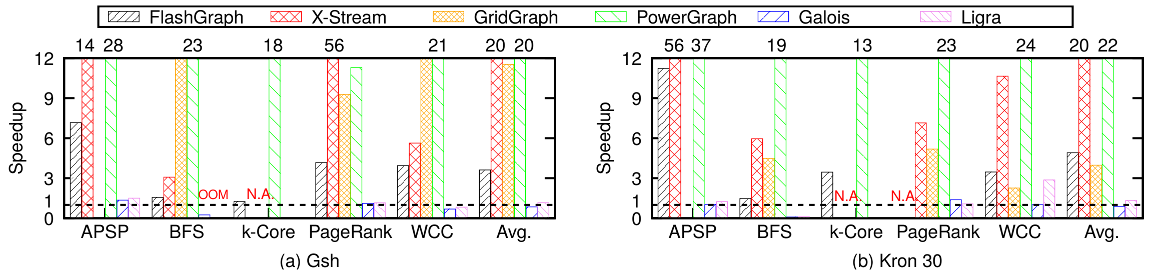
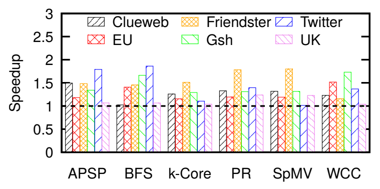


Figure 4.13: Graphene vs. state-of-the-art.

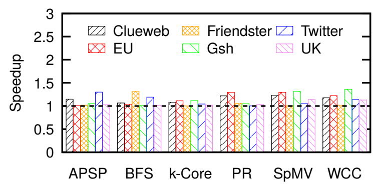
4.8 Evaluations

We have implemented a prototype of Graphene in 3,300 lines of C++ code, where the IoIterator accounts for 1,300 lines and IO functions 800 lines. Six graph algorithms are implemented with average 200 lines of code. We perform our experiments on a server with a dual-socket Intel Xeon E5-2620 processor (total 12 cores and 24 threads with hyperthreading), 128GB memory, 16 500GB Samsung 850 SSDs connected with two LSI SAS 9300-8i host bus adapters, and Linux kernel 4.4.0.

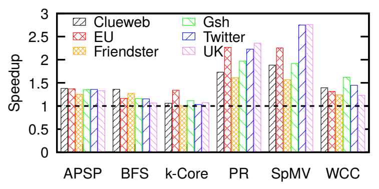
Table 4.2 lists all the graphs used in this paper. Specifically, Twitter [11] and Friendster [193] are real-world social graphs. In particular, Twitter contains 52,579,682 vertices and 1,963,263,821 edges, and Friendster is an online gaming network with 68,349,466 vertices and 2,586,147,869 edges. In addition, Clueweb [203], EU [204], Gsh [205] and UK [206] are webpage based graphs provided by webgraph [53, 207, 208]. Among them, EU is the largest with over one billion of vertices and 90 billion of edges. On the other hand, two Kronecker graphs are generated with the Graph500 genera-



(a) Bitmap vs. Pluglist.



(b) HugePage vs. 4K Page.



(c) Dedicated threads.

Figure 4.14: Overall performance benefits of IO techniques.

tor [83] with scale 30 and 31, which represent the number of vertices as 1 billion (2^{30}) and 2 billion (2^{31}), with number of edges of 32 billion and 1 trillion. This paper, by default uses 8 bytes to represent a vertex ID unless explicitly noted. We run the tests five times and report the average values.

In addition, Table 4.2 presents the time consumption of the preprocessing step of the row-column balanced 2D partition. On average, our partition method takes 50% longer time than the conventional 2D partition method, e.g., preprocessing the largest Kron31 graph takes 916 seconds. Note that except X-Stream, many graph systems, including FlashGraph, GridGraph, PowerGraph, Galois and Ligra, also require similar or longer preprocessing to prepare the datasets. In the following, we report the runtime of graph algorithms, excluding the preprocessing time for all graph systems.

4.8.1 Comparison with the State of the Art

We compare Graphene against FlashGraph (semi-external memory), X-Stream (external memory), GridGraph (external memory), PowerGraph (in-memory), Galois (in-memory), and Ligra (in-memory) when running various algorithms. Figure 4.13 reports the speedup of Graphene over different systems for all five algorithms. SpMV is currently not supported in other systems except our Graphene, and k-Core is only provided by FlashGraph, PowerGraph and Graphene. In the figure the label "NA" indicates lack of support in the system. In this test, we choose one real graph (Gsh) and one synthetic graph (Kron30). Note that Gsh is the largest graph that is supported by in-memory systems. We have observed similar performance on other graphs.

In general, Graphene outperforms external memory systems FlashGraph, GridGraph and X-Stream by 4.3×, 7.8× and 20×, respectively. Compared to in-memory

Name	APSP	BFS	k-Core	PageRank	WCC	SpMV
Kron31	7,233	2,630	318	25,023	3,023	5,706

Table 4.3: Graphene runtime on Kron31 (seconds).

systems PowerGraph, Galois and Ligra where all graph data are stored in DRAM, Graphene keeps the data on SSDs and reads on-demand, outperforming PowerGraph by 21× and achieving a comparable performance with the other two (90% for Galois and 1.1× for Ligra). Excluding BFS which is the most IO intensive and favors in-memory data, Graphene outperforms Galois and Ligra by 10% and 45%, respectively. We also compare Graphene with an emerging Differential Dataflow system [209] and Graphene is able to deliver an order of magnitude speedup on BFS, PageRank and WCC.

For the Gsh graph, as shown in Figure 4.13, Graphene achieves better performance than other graph systems for different algorithms with exceptions for BFS and WCC. For example, for APSP, Graphene outperforms PowerGraph by 29×, Galois by 35%, Ligra by 50%, FlashGraph by 7.2× and X-Stream by 14×. For BFS and WCC, Graphene runs faster than GridGraph, PowerGraph, FlashGraph and X-Stream, but is slower than the two in-memory systems, mostly due to relatively long access latency on SSDs compared to DRAM. Similar performance benefits can also be observed on the syntheic Kron30 graph.

Trillion-edge graph. We further evaluate the performance of Graphene on Kron31 as presented in Table 4.3. On average, all algorithms take around one hour to finish, with the maximum from PageRank of 6.9 hours while k-Core can be completed in 5.3 minutes. To the best of our knowledge, this is among the first attempts to evaluate trillion-edge graphs on a external-memory graph processing system.

4.8.2 Benefits of IO Techniques

This section examines the impacts on the overall system performance brought by different techniques independently, including Bitmap, hugepage, and dedicated IO and computing threads. We run all six algorithms on all six real-world graphs.

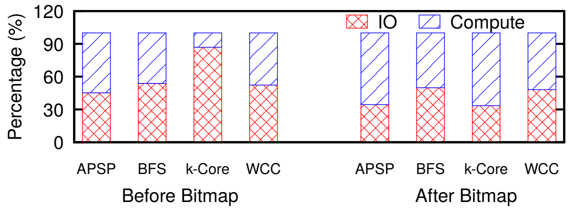


Figure 4.15: Runtime breakdown of IO and computing with Bitmap-based IO.

The Bitmap provides an average 27% improvement over using the pluglist as presented in Figure 4.14(a). Clearly, Bitmap favors the algorithms with massive random IOs such as WCC and BFS and low diameter graphs such as Gsh, EU, and Friendster. For example, Bitmap achieves about 70% speedup on Gsh on both BFS and WCC, and 30% for other algorithms.

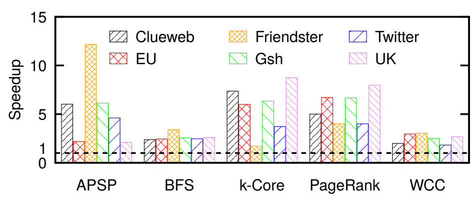
Figure 4.14(b) compares the performance of hugepages and 4KB pages. Hugepages provides average 12% improvement and the speedup varies from 17% for WCC to 6% for k-Core. Again, two largest improvements are achieved on the (largest) Gsh graph for SpMV and WCC.

The benefit introduced by dedicated IO and computing threads is presented in Figure 4.14(c), where the baseline is using one thread for both IO and computing. In this case, Graphene achieves an average speedup of 54%. Particularly, PageRank and SpMV enjoy significant higher improvement (about 2×) than the other algorithms.

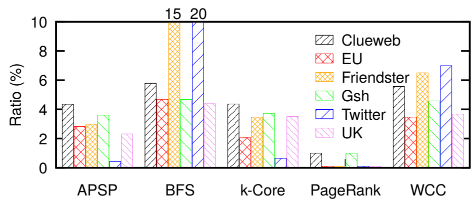
4.8.3 Analysis of Bitmap-based IO

We study how Bitmap-based IO affects the IO and computing ratio of different algorithms in Figure 4.15. Without bitmap, all four algorithms spend about 60% on IO and 40% on computation. In comparison, the distribution of runtime reverses with bitmap, where computation takes average 60% of the time and IO 40%. Because the IO time is significantly reduced, faster IO as a result accelerates the execution of the algorithms. In particular, the biggest change comes from k-Core where IO accounts for 87% and 34% before and after bitmap.

As shown in Figure 4.16, when compared to a pluglist-based approach, the Bitmap-based IO runs 5.5×, 2.6×, 5.6×, 5.7× and 2.5× faster on APSP, BFS, k-Core, PageR-



(a) Preparing Bitmap vs. Pluglist.



(b) Overhead.

Figure 4.16: Bitmap performance and overhead.

ank, and WCC, respectively. Note that here we only evaluate the time consumption of preparing the bitmap and pluglist, which is different from overall system performance presented in Figure 4.14. On the other hand, in most cases, adding Bitmap incurs a small increase of about 3.4% of total IO time. However, for a few cases with relatively high overhead, it is most likely caused by the small size of the graph data (e.g., Friendster and Twitter), as well as random IOs of the algorithms (e.g., BFS). The time spent on Bitmap varies from about 60 milliseconds for PR and SpMV (less than 1% of total IO time), to 100 seconds for APSP (2.3% of IO time).

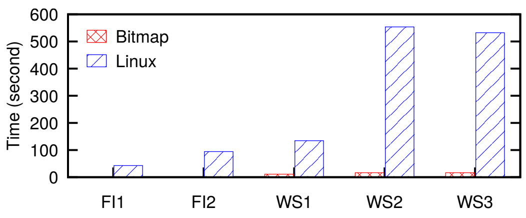
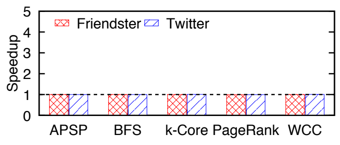
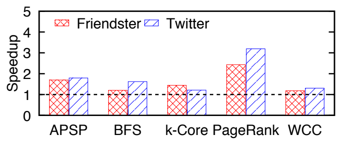


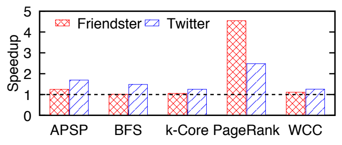
Figure 4.17: Bitmap-based IO performance on traces.



(a) HDD



(b) NVMe



(c) Ramdisk

Figure 4.18: Bitmap performance on HDD, NVMe and Ramdisk.

Bitmap-based IO can be applied to other applications beyond graph processing. Figure 4.17 examines the time consumption differences between Bitmap based IO and Linux IO. Here we replay the reads in five IO traces as quickly as possible, namely Financial 1-2 and WebSearch 1-3 from UMass Trace Repository [210]. On average,

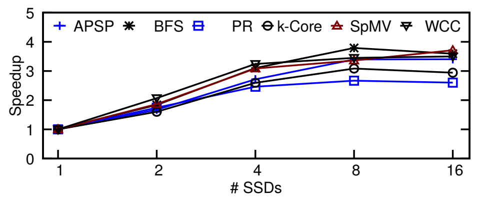


Figure 4.19: Graphene scalability on the Kron30 graph.

the Bitmap is 38× faster than Linux IO, with the maximum speedup of 74× obtained on Financial2 (from 94.2 to 1.26 seconds). The improvement comes mostly from more (9.3×) deduplicated IOs and more aggressive IO merging.

Figure 4.18 further studies the impacts of bitmap based IO on hard disk (HDD), NVMe and Ramdisk. In this test, we use five Seagate 7200RPM SATA III hard drives in a Raid-0 configuration, and one Samsung 950 Pro NVMe device. One can see that compared to the pluglist based method, although bitmap improves hard disk performance only marginally (1% on average), faster storage devices such as NVMe and Ramdisk are able to achieve about 70% improvement in IO performance.

4.8.4 Scalability, Utility, and Throughput

This section studies the scalability of Graphene with respect to the number of SSDs. Recall that Graphene uses two threads per SSD, one IO and another compute. Using a single thread would fail to fully utilize the bandwidth of an SSD. As shown in Figure 4.19, Graphene achieves an average 3.3× speedup on the Kron30 graph when scaling from a single SSD (two threads) to eight SSDs (16 threads). Across different applications, SpMV enjoys the biggest 3.7× speedup and PageRank the smallest 2.6×. The small performance gain from 8 to 16 SSDs is due to the shift of the bottleneck from IO to CPU.

Recall that IO utility is defined as the ratio of useful data and total data loaded, we evaluate the IO utility when using 512-byte IO vs. 4KB IO on various algorithms and graph datasets. As presented in Figure 4.20, Graphene achieves 20% improvement on average. For APSP and BFS, one can see about 30% improvement with the best benefit of 50% on UK. Similar speedups can also be observed for K-Core and WCC.

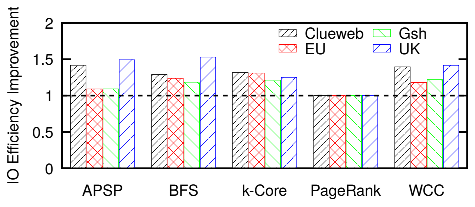


Figure 4.20: Utility of 512-byte vs. 4KB IO.

In contrast, PageRank and SpMV present minimal benefit because the majority of their iterations load the whole graph.

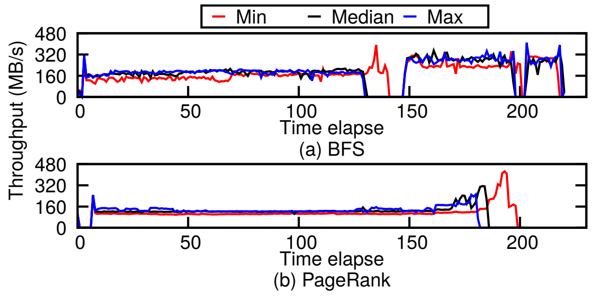


Figure 4.21: Throughputs of the fastest (max) and slowest (min) SSDs, and median throughput out of 16 SSDs.

To demonstrate the IO loads of different disks in Graphene, we further examine the throughput of 16 SSDs for two applications, BFS and PageRank. Figure 4.21 show the throughput for the fastest (max) and slowest (min) SSDs, as well as the median throughput. Clearly, the 16 SSDs are able to deliver similar IO performance for most of run, with an average difference of 6 to 15 MB/s (5-7% for PageRank and BFS). For both algorithms, the slowest disk does require extra time to complete the processing, which we leave for future research to close the gap.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this dissertation, we develop Enterprise, a new GPU-based BFS system, that produces over 70 billion TEPS on a single GPU and 122 billion TEPS on two GPUs, delivering 446 million TEPS per Watt. This is achieved by efficient management of numerous GPU streaming processors and unique memory hierarchy. As part of future work, we plan to integrate Enterprise with high-speed storage and networking devices and run on even larger graphs. We further present iBFS, a new GPU-based concurrent BFS system which leverages a novel GroupBy strategy, combined with joint frontier queue and bitwise operations, to achieve high-performance concurrent breadth-first traversals. iBFS achieves unprecedented performance of over 57,000 billion TEPS on over 100 GPUs. As part of future work, we plan to explore additional optimizations for iBFS and study its application on a wide range of domains.

This dissertation further proposes SIMD-X, a parallel graph computing framework that supports programming and processing of single instruction multiple, complex, data on GPUs. Specifically, the Active-Compute-Combine (ACC) model provides ease of programming to programmers, while just-in-time task management and push-pull based kernel fusion leverage the opportunities for system-level optimization. Using SIMD-X, a user can program a graph algorithm in tens of lines of code, while achieving upto an order of magnitude speedup compared to the state-of-the-art [4, 211, 212].

Last but not the least, our Graphene – a big graph analytics system – consists of a number of novel techniques including IO centric processing, Bitmap-based asynchronous IO, hugepage support, data and workload balancing. It allows the users to treat the data as in-memory, while delivering high-performance on SSDs. The experiments show that Graphene is able to perform comparably against in-memory processing systems on large-scale graphs, and also runs several times faster than existing external-memory processing systems.

5.2 Future Work

Graph computation becomes a hot topic recently and many challenges await researchers. For instance, graph challenge [213] unveils an aggressive goal: accelerating graph computation by 1,000×. Though my previous research can achieve one order of magnitude speedup, it is still far from 1,000×.

My future research will center around further improving the performance of graph analytics through hardware and software co-designed platforms. Particularly, neither moving data from storage media to CPU in off-the-shelf hardware nor dispatching instructions to storage media as proposed in Processing In Memory (PIM) is enough to fully accelerate graph analytics. We admit PIM can partially address the data movement problem, but it faces several remaining challenges, e.g., balancing the data mapping across different PIM cells, and addressing heavy communication overhead across PIM units. As a result, I plan to (1) enhance my existing row-column balanced partitioning method to help balance the workload distribution for PIM architecture; (2) through my expertise in graph algorithms, improve the hardware design of PIM, e.g., graph-aware atomic operation support and diverse page sizes; (3) design better software layers for PIM hardware that could fully exploit hardware potential and expose easy programming interfaces to end user.

Bibliography

[1] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpu servers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[2] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[3] H. Liu and H. H. Huang, "Simd-x: Programming and processing of graph algorithms on gpus," in *Under submission*, 2017.

[4] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[5] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017.

[6] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 456–471.

[7] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *WWW*, 2010.

[8] D. Levinthal, "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors," *Intel Performance Analysis Guide*, 2009.

[9] NVIDIA Corporation, "Nvidia cuda c programming guide," 2011.

[10] Sandisk SSD VS. HDD, "https://www.sandisk.com/content/dam/sandisk-main/en_us/assets/resources/enterprise/infographics/how-do-ssds-stack-up-against-hdds.pdf," 2016.

[11] Twitter (MPI) Network Dataset – KONECT, "http://konect.uni-koblenz.de/networks/twitter_mpi," 2016.

[12] Facebook Friend Recommendation Algorithm, "https://techcrunch.com/2017/06/27/facebook-2-billion-users/k."

[13] H. H. Huang and H. Liu, "Big data machine learning and graph analytics: Current state and future challenges," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 16–17.

[14] B. Zheng, H. Su, W. Hua, K. Zheng, X. Zhou, and G. Li, "Efficient clue-based route search on road networks," *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[15] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. Sadiq, and X. Zhou, "Approximate keyword search in semantic trajectory database," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 975–986.

[16] B. Zheng, H. Wang, K. Zheng, H. Su, K. Liu, and S. Shang, "Sharkdb: an in-memory column-oriented storage for trajectory analysis," *World Wide Web*, pp. 1–31, 2017.

[17] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A comparative analysis of data center network architectures," in *2014 IEEE International Conference on Communications (ICC)*. IEEE, 2014, pp. 3106–3111.

[18] J. Wu, S. Subramaniam, and H. Hasegawa, "Optimal nonuniform wavebanding in wdm mesh networks," *Photonic Network Communications*, vol. 31, no. 3, pp. 376–385, 2016.

[19] J. Wu, M. Xu, S. Subramaniam, and H. Hasegawa, "Routing, fiber, band, and spectrum assignment (rfbsa) for multi-granular elastic optical networks," in *Communications (ICC), 2017 IEEE International Conference on.* IEEE, 2017, pp. 1–6.

[20] J. Wu, J. Zhao, and S. Subramaniam, "Co-scheduling computational and networking resources in elastic optical networks," in *Communications (ICC), 2014 IEEE International Conference on.* IEEE, 2014, pp. 3307–3312.

[21] C. Liu, M. Xu, and S. Subramaniam, "A reconfigurable high-performance optical data center architecture," in *Global Communications Conference (GLOBECOM), 2016 IEEE.* IEEE, 2016, pp. 1–6.

[22] M. Xu, C. Liu, and S. Subramaniam, "Podca: A passive optical data center architecture," in *Communications (ICC), 2016 IEEE International Conference on.* IEEE, 2016, pp. 1–6.

[23] Y. Xiang, H. Liu, T. Lan, H. Huang, and S. Subramaniam, "Optimizing job reliability via contention-free, distributed scheduling of vm checkpointing," in *Proceedings of the ACM SIGCOMM workshop on Distributed cloud computing*, 2014.

[24] Y. Xiang, H. Liu, T. Lan, H. Huang, and S. Subramaniam, "Optimizing job reliability through contention-free, distributed checkpoint scheduling," Tech. Rep.

[25] G. Song, M. Chao, B. Yang, and Y. Zheng, "Tlr: A traffic-light-based intelligent routing strategy for ngeo satellite ip networks," *IEEE Transactions on Wireless Communications*, vol. 13, no. 6, pp. 3380–3393, June 2014.

[26] G. SONG, M. CHAO, B. YANG, H. ZHONG, and Y. ZHENG, "Study on multipath qos routing strategy in satellite networks [j]," *Journal of Spacecraft TT&C Technology*, vol. 6, p. 004, 2012.

[27] G. Song, M. Chao, B. Yang, H. Zhong, and Y. Zheng, "Research on multipath qos routing strategy for the satellite network," in *Proceedings of the 26th Conference of Spacecraft TT&C Technology in China*. Springer, 2013, pp. 289–298.

[28] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani, "Statsym: vulnerable path discovery through statistics- guided symbolic execution," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[29] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "Simber: Eliminating redundant memory bound checks via statistical inference," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, Cham, 2017, pp. 413–426.

[30] Y. Zhao, Y. Jian, Z. Liu, H. Liu, Q. Liu, C. Chen, Z. Li, L. Wang, H. H. Huang, and C. Zeng, "Network analysis reveals the recognition mechanism for dimer formation of bulb-type lectins," in *Nature Scientific Report*, 2017.

[31] Y. Jian, Y. Zhao, and C. Zeng, "Network analysis reveals the recognition mechanism for complex formation of mannose-binding lectins," *Bulletin of the American Physical Society*, vol. 62, 2017.

[32] H. Liu, J.-H. Seo, R. Mittal, and H. H. Huang, "Matrix decomposition based conjugate gradient solver for poisson equation," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1499–1500.

[33] H. Liu, J.-H. Seo, R. Mittal, and H. H. Huang, "Gpu-accelerated scalable solver for banded linear systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[34] R. Mittal, J. H. Seo, V. Vedula, Y. J. Choi, H. Liu, H. H. Huang, S. Jain, L. Younes, T. Abraham, and R. T. George, "Computational modeling of cardiac hemodynamics: current status and future outlook," *Journal of Computational Physics*, vol. 305, pp. 1065–1082, 2016.

[35] F. Li, T. Xu, D.-H. T. Nguyen, X. Huang, C. S. Chen, and C. Zhou, "Label-free evaluation of angiogenic sprouting in microengineered devices using ultrahigh-resolution optical coherence microscopy," *Journal of biomedical optics*, vol. 19, no. 1, pp. 016 006–016 006, 2014.

[36] F. Li, Y. Song, A. Dryer, W. Cogguillo, Y. Berdichevsky, and C. Zhou, "Nondestructive evaluation of progressive neuronal changes in organotypic rat hippocampal slice cultures using ultrahigh-resolution optical coherence microscopy," *Neurophotonics*, vol. 1, no. 2, pp. 025 002–025 002, 2014.

[37] C.-K. Yeh, N. Matsuda, X. Huang, F. Li, M. Walton, and O. Cossairt, "A streamlined photometric stereo framework for cultural heritage," in *European Conference on Computer Vision*. Springer, 2016, pp. 738–752.

[38] T. Xu, F. Li, D.-H. T. Nguyen, C. S. Chen, C. Zhou, and X. Huang, "Delineating 3d angiogenic sprouting in oct images via multiple active contours," in *Augmented Reality Environments for Medical Imaging and Computer-Assisted Interventions*. Springer, 2013, pp. 231–240.

[39] F. Li, N. Matsuda, M. Walton, and O. Cossairt, "Fluorescence lifetime estimation using a dynamic vision sensor," in *SPIE Commercial+ Scientific Sensing and Imaging*. International Society for Optics and Photonics, 2017, pp. 102 220N–102 220N.

[40] Gnutella, "https://sourceforge.net/projects/gtk-gnutella/."

[41] Large Networks Visualization Tool (Lanet-vi), "http://lanet-vi.fi.uba.ar/."

[42] J. Gräßler, D. Koschützki, and F. Schreiber, "Centilib: comprehensive analysis and exploration of network centralities," *Bioinformatics*, 2012.

[43] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[44] Google Map, "https://developers.google.com/maps/premium/."

[45] simFlow: Comprehensive CFD software, "https://sim-flow.com/."

[46] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO)*, 2004.

[47] Facebook Friend Recommendation Algorithm, "https://www.quora.com/How-does-Facebooks-friend-recommendation-system-work."

[48] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[49] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, "The more the merrier: Efficient multi-source graph traversal," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 2014.

[50] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.

[51] P. Kumar and H. H. Huang, "G-store: High-performance graph store for trillion-edge processing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[52] P. Kumar and H. H. Huang, "Falcon: Scaling io performance in multi-ssd volumes," in *USENIX Annual Technical Conference (USENIX ATC).* USENIX Association, 2017, pp. 41–53.

[53] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW)*, Manhattan, USA, 2004.

[54] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *Data Compression Conference (DCC), 2015.* IEEE, 2015, pp. 403–412.

[55] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High performance computing–HiPC 2007.* Springer, 2007, pp. 197–208.

[56] H. Yanagisawa, "A multi-source label-correcting algorithm for the all-pairs shortest paths problem," in *International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2010, pp. 1–10.

[57] I.-L. Wang, E. L. Johnson, and J. S. Sokol, "A multiple pairs shortest path algorithm," *Transportation science*, vol. 39, no. 4, pp. 465–476, 2005.

[58] R. Seidel, "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of computer and system sciences*, vol. 51, no. 3, pp. 400–403, 1995.

[59] A. Sarje and S. Aluru, "All-pairs computations on many-core graphics processors," *Parallel Computing*, vol. 39, no. 2, pp. 79–93, 2013.

[60] U. Meyer and P. Sanders, "δ-stepping: A parallel single source shortest path algorithm," *Algorithms—ESA'98*, 1998.

[61] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of Interna-*

tional Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013.

[62] G. M. Slota, S. Rajamanickam, and K. Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[63] Nvidia, "Nvidia kepler gk110 architecture whitepaper," 2013.

[64] Nvidia Profiler Tools, "http://docs.nvidia.com/cuda/profiler-users-guide/."

[65] Green Graph500, "http://green.graph500.org/."

[66] J. Chen, F. Yao, and G. Venkataramani, "Watts-inside: A hardware-software cooperative approach for multicore power debugging," in *31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 335–342.

[67] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A dual delay timer strategy for optimizing server farm energy," in *7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 258–265.

[68] F. Yao, J. Wu, S. Subramaniam, and G. Venkataramani, "Wasp: Workload adaptive energy-latency optimization in server farms using server low-power states," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[69] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, 2004.

[70] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 1–10.

[71] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *PPoPP*, 2013.

[72] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[73] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *PPoPP*, 2011.

[74] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *PPoPP*, 2012.

[75] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2015, pp. 265–266.

[76] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies.* USENIX Association, 2015, pp. 45–58.

[77] S. Dolev, Y. Elovici, and R. Puzis, "Routing betweenness centrality," *Journal of the ACM (JACM)*, vol. 57, no. 4, p. 25, 2010.

[78] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2009, pp. 1–8.

[79] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* IEEE, 2014, pp. 572–583.

[80] V. Ufimtsev and S. Bhowmick, "Application of group testing in identifying high betweenness centrality vertices in complex networks," in *11th Workshop on Machine Learning with Graphs, KDD*, 2013.

[81] P. W. Olsen, A. G. Labouseur, and J.-H. Hang, “Efficient top-k closeness centrality search,” in *International Conference on Data Engineering (ICDE).* IEEE, 2014, pp. 196–207.

[82] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Regularizing graph centrality computations,” *Journal of Parallel and Distributed Computing*, 2014.

[83] Graph500, “http://www.graph500.org/.”

[84] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, “The large-scale organization of metabolic networks,” *Nature*, vol. 407, no. 6804, pp. 651–654, 2000.

[85] R. Albert, H. Jeong, and A.-L. Barabási, “Internet: Diameter of the world-wide web,” *Nature*, vol. 401, no. 6749, pp. 130–131, 1999.

[86] B. A. Huberman and L. A. Adamic, “Internet: Growth dynamics of the world-wide web,” *Nature*, 1999.

[87] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world.* Cambridge University Press, 2010.

[88] Y. Ji, Y. He, X. Jiang, J. Cao, and Q. Li, “Combating the evasion mechanisms of social bots,” *computers & security*, vol. 58, pp. 230–249, 2016.

[89] J. Cao, Q. Li, Y. Ji, Y. He, and D. Guo, “Detection of forwarding-based malicious urls in online social networks,” *International Journal of Parallel Programming*, vol. 44, no. 1, pp. 163–180, 2016.

[90] Y. Ji, Y. He, X. Jiang, and Q. Li, “Towards social botnet behavior detecting in the end host,” in *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on.* IEEE, 2014, pp. 320–327.

[91] R. Nasre, M. Burtscher, and K. Pingali, “Data-driven versus topology-driven irregular computations on gpus,” in *IPDPS*, 2013.

[92] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 2011.

[93] D. Li and M. Becchi, "Deploying graph algorithms on gpus: An adaptive solution," in *IPDPS*, 2013.

[94] U. Brandes, "A faster algorithm for betweenness centrality*," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[95] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: Scalable reachability index for large graphs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 276–284, 2010.

[96] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu, "K-reach: who is in your small world," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1292–1303, 2012.

[97] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *Proceedings of the SIGMOD International Conference on Management of data.* ACM, 2009, pp. 813–826.

[98] S. Mahajan and J. Malhotra, "Energy efficient path determination in wireless sensor network using bfs approach," *Wireless Sensor Network*, vol. 3, no. 11, p. 351, 2011.

[99] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *nature*, vol. 406, no. 6794, pp. 378–382, 2000.

[100] M. Bosse, P. Newman, J. Leonard, and S. Teller, "Simultaneous localization and map building in large-scale cyclic environments using the atlas framework," *The International Journal of Robotics Research*, vol. 23, no. 12, pp. 1113–1139, 2004.

[101] S. Ichikawa, "Navigation system and method for calculating a guide route," Feb. 26 2002, uS Patent 6,351,707.

[102] A. Bonifati, R. Ciucanu, and A. Lemay, "Learning path queries on graph databases," in *18th International Conference on Extending Database Technology (EDBT)*, 2014.

[103] M. Najork and J. L. Wiener, "Breadth-first crawling yields high-quality pages," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 114–118.

[104] X. Lu, T. Q. Phan, and S. Bressan, "Incremental algorithms for sampling dynamic graphs," in *Database and Expert Systems Applications*. Springer, 2013, pp. 327–341.

[105] A. E. Sariyuce, E. Saule, K. Kaya, and U. V. Catalyurek, "Hardware/software vectorization for closeness centrality on multi-/many-core architectures," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2014, pp. 1386–1395.

[106] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 52–55.

[107] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC*, 2009.

[108] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *GH*, 2008.

[109] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010.

[110] A. Cohen, T. Grosser, P. H. Kelly, J. Ramanujam, P. Sadayappan, S. Verdoolaege *et al.*, "Split tiling for gpus: Automatic parallelization using trapezoidal tiles to reconcile parallelism and locality, avoiding divergence and load imbalance," in *GPGPU*, 2013.

[111] Z. Qi, Y. Xiao, B. Shao, and H. Wang, "Toward a distance oracle for billion-node graphs," *VLDB*, 2013.

[112] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[113] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *IPDPS*, 2013.

[114] P. Klodt, G. Weikum, S. Bedathur, and S. Seufert, "Indexing strategies for constrained shortest paths over large social networks," Ph.D. dissertation, 2011.

[115] H. Cao, K. S. Candan, and M. L. Sapino, "Skynets: Searching for minimum trees in graphs with incomparable edge weights," in *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2011, pp. 1775–1784.

[116] J. Gao, J. X. Yu, R. Jin, J. Zhou, T. Wang, and D. Yang, "Neighborhood-privacy protected shortest distance computing in cloud," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 409–420.

[117] W. Yu, X. LIN, W. Zhang, J. McCann *et al.*, "Fast all-pairs simrank assessment on large graphs and bipartite domains," 2014.

[118] A. D. Zhu, X. Xiao, S. Wang, and W. Lin, "Efficient single-source shortest path and distance queries on large graphs," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 998–1006.

[119] J. Zhong and B. He, "Parallel graph processing on graphics processors made easy," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1270–1273, 2013.

[120] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1543–1552, 2014.

[121] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* IEEE Computer Society, 2010, pp. 1–11.

[122] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *28th International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2014, pp. 349–359.

[123] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "Phast: Hardware-accelerated shortest path trees," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 940–952, 2013.

[124] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *27th International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2013, pp. 548–559.

[125] P. N. Klein, "Multiple-source shortest paths in planar graphs," in *SODA*, vol. 5, 2005, pp. 146–155.

[126] S. Cabello, E. W. Chambers, and J. Erickson, "Multiple-source shortest paths in embedded graphs," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1542–1571, 2013.

[127] C.-L. Yang, H.-W. Tseng, C.-C. Ho, and J.-L. Wu, "Software-controlled cache architecture for energy efficiency," *Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 634–644, 2005.

[128] P.-H. Wang, Y.-M. Chen, C.-L. Yang, and Y.-J. Cheng, "A predictive shutdown technique for gpu shader processors," *Computer Architecture Letters*, 2009.

[129] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *GPGPU*, 2013.

[130] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," UVA, Tech. Rep., 2009.

[131] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU gems*, 2007.

[132] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[133] Intel Inc., "Product brief intel xeon processor e7-8800/4800/2800 v2 product families," 2014.

[134] Intel Inc., "Hpc code modernization workshop at lrz," 2015.

[135] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Practical Recommendations on Crawling Online Social Networks," *JSAC*, 2011.

[136] The University of Florida: Sparse Matrix Collection, "http://www.cise.ufl.edu/research/sparse/matrices/."

[137] SNAP: Stanford Large Network Dataset Collection., "http://snap.stanford.edu/data/."

[138] GTgraph: A suite of synthetic random graph generators, "http://www.cse.psu.edu/~madduri/software/GTgraph/."

[139] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems.* ACM, 2014, pp. 1–6.

[140] GraphBIG, "https://github.com/graphbig/graphBIG."

[141] S. Bressan, A. Cuzzocrea, P. Karras, X. Lu, and S. H. Nobari, "An effective and efficient parallel approach for random graph generation over gpus," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 303–316, 2013.

[142] B. R. Gaster and L. Howes, "Can gpgpu programming be liberated from the data-parallel bottleneck?" *Computer*, 2012.

[143] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing.* ACM, 2014, pp. 239–252.

[144] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound gpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2014.

[145] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 135–146.

[146] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," 2010.

[147] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation.* USENIX Association, 2012, pp. 31–46.

[148] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15).* USENIX Association, 2015, pp. 375–386.

[149] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of international conference on Knowledge discovery and data mining (SIGKDD)*, 2013, pp. 77–85.

[150] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2013, pp. 505–516.

[151] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 2013, pp. 472–488.

[152] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 40, no. 1, 2012, pp. 349–362.

[153] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 2015, pp. 410–424.

[154] K. Wang and Z. Su, "Graphq: Graph query processing with abstraction refinement-scalable and programmable analytics over very large graphs on a single pc."

[155] D. Li, "Facilitating emerging applications on many-core processors," Ph.D. dissertation, University of Missouri–Columbia, 2016.

[156] D. Li, X. Chen, M. Becchi, and Z. Zong, "Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus," in *International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom)*, 2016.

[157] D. Li, S. Chakradhar, and M. Becchi, "Grapid: A compilation and runtime framework for rapid prototyping of graph applications on many-core processors," in *20th International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.

[158] H. Wu, D. Li, and M. Becchi, "Compiler-assisted workload consolidation for efficient dynamic parallelism on gpu," in *International Parallel and Distributed Processing Symposium*, 2016.

[159] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, 2013.

[160] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[161] S. Yan, G. Long, and Y. Zhang, "Streamscan: fast scan algorithms for gpus without global barrier synchronization," in *PPoPP*, 2013.

[162] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[163] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[164] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Practical recommendations on crawling online social networks," *IEEE Journal on Selected Areas in Communications*, 2011.

[165] European Open Stream Map, "http://download.geofabrik.de/europe-latest.osm.bz2,."

[166] NVIDIA Tesla P100 GPU Accelerator, "https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf."

[167] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: Gpu graph analytics," *arXiv preprint arXiv:1701.01170*, 2017.

[168] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, 2004.

[169] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, 2001.

[170] A. Del Sol, H. Fujihashi, and P. O'Meara, "Topology of small-world networks of protein-protein complex structures," *Bioinformatics*, 2005.

[171] C. Doerr and N. Blenn, "Metric convergence in social network sampling," in *Proceedings of the 5th ACM workshop on HotPlanet*, 2013.

[172] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the european conference on Computer systems (Eurosys)*, 2014.

[173] Y. Ji, Q. Li, Y. He, and D. Guo, "Botcatch: leveraging signature and behavior for bot detection," *Security and Communication Networks*, vol. 8, no. 6, pp. 952–969, 2015.

[174] Y. Ji, Y. He, D. Zhu, Q. Li, and D. Guo, "A mulitiprocess mechanism of evading behavior-based bot detection approaches." in *ISPEC*.

[175] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 2015, pp. 425–440.

[176] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[177] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles (SOSP)*, 2013.

[178] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *USENIX Annual Technical Conference (ATC)*, 2016.

[179] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002.

[180] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Faking the pulse of a fast-changing and connected world," in *Proceedings of the european conference on Computer Systems (Eurosys)*, 2012.

[181] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[182] P. Kumar and H. H. Huang, "G-store: High-performance graph store for trillion-edge processing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[183] Y. Xiang, T. Lan, V. Aggarwal, and Y. F. R. Chen, "Joint latency and cost optimization for erasurecoded data center storage," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 2, pp. 3–14, 2014.

[184] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012.

[185] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[186] D. Bovet and M. Cesati, *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[187] L. Cui, J. Li, B. Li, J. Huai, C. Hu, T. Wo, H. Al-Aqrabi, and L. Liu, "Vmscatter: Migrate virtual machines to many hosts," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 63–72.

[188] L. Cui, B. Li, Y. Zhang, and J. Li, "Hotsnap: A hot distributed snapshot system for virtual machine cluster." in *LISA*, 2013, pp. 59–74.

[189] L. Cui, J. Li, T. Wo, B. Li, R. Yang, Y. Cao, and J. Huai, "Hotrestore: A fast restore system for virtual machine cluster." in *LISA*, 2014, pp. 1–16.

[190] L. Cui, Z. Hao, Y. Peng, and X. Yun, "Piccolo: A fast and efficient rollback system for virtual machine clusters," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[191] L. Cui, T. Wo, B. Li, J. Li, B. Shi, and J. Huai, "Pars: A page-aware replication system for efficiently storing virtual machine snapshots," in *ACM SIGPLAN Notices*, vol. 50, no. 7. ACM, 2015, pp. 215–228.

[192] J. Li, H. Liu, L. Cui, B. Li, and T. Wo, "irow: An efficient live snapshot system for virtual machine disk," in *18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012, pp. 376–383.

[193] Friendster Network Dataset – KONECT, "http://konect.uni-koblenz.de/networks/friendster," 2016.

[194] Samsung 850 EVO SSD, "http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850evo.html," 2015.

[195] Fixing asynchronous I/O, again, "https://lwn.net/Articles/671649/," 2016.

[196] Performance Issues with Transparent Huge Pages (THP), "https://blogs.oracle.com/linux/entry/performance_issues_with_transparent_huge," 2013.

[197] Transparent huge pages in 2.6.38, "http://lwn.net/Articles/423584/," 2011.

[198] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth Symposium on Cloud Computing (SoCC)*, 2015.

[199] X. Ding, K. Wang, and X. Zhang, "Ulcc: A user-level facility for optimizing shared cache performance on multicores," in *Proceedings of the SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2011.

[200] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the European conference on Computer systems (Eurosys)*, 2009.

[201] T. H. Haveliwala, "Topic-sensitive pagerank," in *Proceedings of the 11th international conference on World Wide Web (WWW)*, 2002.

[202] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, 2013.

[203] Clueweb dataset from WebGraph, "http://law.di.unimi.it/webdata/clueweb12/," 2012.

[204] EU dataset from WebGraph, "http://law.di.unimi.it/webdata/eu-2015/," 2015.

[205] Gsh dataset from WebGraph, "http://law.di.unimi.it/webdata/gsh-2015/," 2015.

[206] UK dataset in WebGraph, "http://law.di.unimi.it/webdata/uk-2014/," 2014.

[207] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web (WWW)*, 2011.

[208] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web (WWW)*, 2014.

[209] Timely Dataflow Blog, "https://github.com/frankmcsherry/timely-dataflow," 2016.

[210] UMASS Trace Repository, "http://traces.cs.umass.edu/," 2016.

[211] Y. Wang, S. Baxter, and J. D. Owens, "Mini-gunrock: A lightweight graph analytics framework on the GPU," in *Graph Algorithms Building Blocks*, ser. GABB 2017, May 2017. [Online]. Available: https://escholarship.org/uc/item/5wm061tr

[212] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," in *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2017, May/Jun. 2017. [Online]. Available: http://escholarship.org/uc/item/39r145g1

[213] DARPA Graph Challenge, "https://graphchallenge.mit.edu/darpa-hive."