Gunrock: A Programming Model and Implementation for Graph Analytics on Graphics Processing Units

By

Yangzihao Wang
B.Eng. (Beihang University) 2007
M.Eng. (Beihang University) 2011

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

John D. Owens, Chair

Nina Amenta

Zhaojun Bai

Committee in Charge

2016

To my parents, Xinhua and Kunxia; my wife Wenjing and our daughter Ava …

Contents

List of Figures

List of Tables

Abstract

Gunrock: A Programming Model and Implementation for Graph Analytics on Graphics Processing Units

The high-performance, highly parallel, fully programmable modern Graphics Processing Unit's high memory bandwidth, computing power, excellent peak throughput, and energy efficiency brings acceleration to regular applications that have extensive data parallelism, regular memory access patterns, and modest synchronizations. However, for graph analytics, the inherent irregularity of graph data structures leads to irregularity in data access and control flow, making efficient graph analytics on GPUs a significant challenge. Despite some promising specialized GPU graph algorithm implementations, parallel graph analytics on the GPU in general still faces two major challenges. The first is the programmability gap between low-level implementations of specific graph primitives and a general graph processing system. Programming graph algorithms on GPUs is difficult even for the most skilled programmers. Specialized GPU graph algorithm implementations do not generalize well since they often couple a specific graph computation to a specific type of parallel graph operation. The second is the lack of a GPU-specific graph processing programming model. High-level GPU programming models for graph analytics often recapitulate CPU programming models and do not compare favorably in performance with specialized implementations due to different kinds of overhead introduced by maintaining a high-level framework.

This dissertation seeks to resolve the conflict of programmability and performance for graph analytics on the GPU by designing a GPU-specific graph processing programming model and building a graph analytics system on the GPU that not only allows quick prototyping of new graph primitives but also delivers the performance of customized, complex GPU hardwired graph primitives. To achieve this goal, we present a novel data-centric abstraction for graph operations that allows programmers to develop graph primitives at a high level of abstraction while simultaneously delivering high performance by incorporating several profitable optimizations, which previously were only applied to different individual graph algorithm implementations on the

GPU, into the core of our implementation, including kernel fusion, push-pull traversal, idempotent traversal, priority queues, and various workload mapping strategies. We design and implement a new graph analytics system, Gunrock, which contains a set of simple and flexible graph operation APIs that can express a wide range of graph primitives at a high level of abstraction. Using Gunrock, we implement a large set of graph primitives, which span from traversal-based algorithms and ranking algorithms, to triangle counting and bipartite-graph-based algorithms. All of our graph primitives achieve comparable performance to their hardwired counterparts and significantly outperform previous programmable GPU abstractions.

Acknowledgments

First and foremost, I want to thank my advisor, John D. Owens. Joining John's research group forever changed my life (in a good way). John introduced me to general purpose parallel computing, taught me knowledge of this research field, showed me how to do two of the most important things in research: raising the right questions and making your research work impactful. He gave me freedom and encouragement to explore different research topics, pursue my own ideas, do many internships, and attend many conferences. Through my entire PhD life, I have always had his unfailing help and support on everything, be it academic related, or personal issue. It is just wonderful to work with him.

I would also like to thank the rest of my dissertation committee and my qualifying exam committee members for helping me throughout my graduate studies. Both Nina Amanta and Zhaojun Bai have provided excellent feedback on my research and I learned so much from their courses on parallel computing and large-scale scientific computing. Bernd Hamann and Felix Wu provided lots of helps and feedback during the preparation of my qualifying exam. It is my honor to have such esteemed researchers evaluate my work. Special thanks to Yong Jae Lee and students from his research group. Yong Jae raised my interest of deep learning in his wonderful deep learning seminar. He and his students: Fanyi Xiao, Krishna Kumar Singh, Maheen Rashid, and Zhongzheng Ren, have taught me a lot through several paper reading sessions.

Other than my advisor, several people's ideas have also inspired this dissertation, I am thankful to: Duane Merrill, Sean Baxter, Oded Green, Julian Shun, Scott Beamer, and Adam McLaughlin, for the inspirations they have brought to me through either emails or in-person technical discussions. Especially Duane Merrill, Sean Baxter, and Julian Shun. Gunrock's programming style is heavily influenced by their wonderful works (b40c, CUB, Ligra, and moderngpu).

During my graduate studies, I also had the opportunity to interact with many researchers and domain experts from different organizations and companies. I want to thank Michael Garland for inviting me to give a talk at NVIDIA Research, and Duane Merrill for spending an afternoon with me debugging my code and giving me a tour at University of Virginia. I would also like to

thank Nikolai Sakharnykh and Steven Dalton, for providing several technical details regarding CUDA, matrix and memory optimizations on the GPU.

I would like to thank my colleagues at Onu Technology Inc.: Erich Elsen, Guha Jayachandran and Vishal Vaidyanathan for their great work on VertexAPI2 and helps on Gunrock. It has always been a delight working with them. I want to thank Oded Green from Georgia Tech, who helped improve Gunrock by providing both technical insights and code patches. I want to thank Sungpack Hong and Hassan Chafi from Oracle Research for inviting me to give a talk and providing feedback on dataset and graph primitive picking.

I was fortunate to work on DARPA's XDATA project. I would like to thank our DARPA program managers Wade Shen and Christopher White, DARPA business manager Gabriela Araujo, and our principal investigator, Eric Whyne. Their support and help is crucial for me to smoothly finish my research. I would also like to thank several colleagues from XDATA projects: Peter Wang from Continuum, Curtis Lisle from KnowledgeVis, LLC, and Ryan Hafen. My collaboration with them helped me gain knowledge on a wide range of technology stacks spanning from Python, data visualization, to R.

During my two internships, I learned both technical insights of the industry and how to work efficiently as a software engineer. I want to thank my mentors: Benedict Gaster (AMD) and Ivan Yulaev (Google).

My research would not have been finished if not for the help of the following funding agencies: DARPA XDATA program under US Army award W911QX-12-C-0059 and DARPA STTR awards D14PC00023 and D15PC00010; NSF awards CCF-1017399, OCI-1032859, and CCF-1629657; and UC Lab Fees Research Program Award 12-LR-238449.

I spent most of my time doing research in our lab. I hereby thank all my fellow Aggies: Calina Copos, Afton Geil, Jason Mak, Kerry Seitz, Leyuan Wang, Yuechao Pan, Vehbi Eşref Bayraktar, Saman Ashkiani, Collin McCarthy, Andy Riffel, Carl Yang, Weitang Liu, Muhammand Osama, Chenshan Shari Yuan, Shalini Venkataraman, Ahmed H. Mahmoud, Muhammand Awad, and Yuxin Chen. You have made our lab a place full of happiness and creativity. I really enjoyed

studying, working, and having fun with you guys, you guys will surely be missed!

I would also like to thank the alumni of our lab who have spent good times with me before they graduated: Anjul Patney, Stanley Tzeng, Andrew Davidson, Jeff Sturat, Pinar Muyan-Özçelik, Ritesh Patel, Yao Zhang, and Yuduo Wu.

I especially want to thank all members of team Gunrock who have worked with me on this project. Gunrock has always been the result of group efforts. I cannot ask for a better team to work with!

My life would be much more boring without these good friends in Davis: I would like to thank Teng Wang, Sam Siu, Keith Wang, Huan Zhang, and Zou You.

Last but not least, I wish to thank my family. My parents are always unconditionally encouraging and supportive for every decision I made along my academic journey. Without their wholehearted support, I would not have come this far. My special thanks and love to my wife, Wenjing Pan, there is so much to thank her for. It is only with her love, support, and patience, that I can fully focus on my research and pursue my dream. My life is meaningful with her accompany. Finally, I wish to express my love and gratitude to my lovely daughter Ava. Her birth is the best thing that has ever happened to my wife and I. I could not imagine a better way to end my PhD than having little Ava around me.

Chapter 1

Introduction

Graphs are ubiquitous data structures that can represent relationships between people (social networks), computers (the Internet), biological and genetic interactions, and elements in unstructured meshes. Many practical problems in social networks, physical simulations, bioinformatics, and other applications can be modeled in their essential form by graphs and solved with appropriate graph primitives. Various types of such graph primitives that compute and exploit properties of particular graphs are collectively known as graph analytics. In the past decade, as graph problems grow larger in scale and become more computationally complex, the research of parallel graph analytics has raised great interest to overcome the computational resource and memory bandwidth limitations of single processors.

Modern Graphics Processing Units (GPUs) are high-performance, highly parallel, fully programmable architectures with high memory bandwidth and computing power. [1] Their excellent peak throughput and energy efficiency brings acceleration to regular applications that have extensive data parallelism, regular memory access patterns, and modest synchronizations [41].

For graph analytics, the inherent irregularity of graph data structures leads to irregularity in data access and control flow, making efficient graph analytics on GPUs a significant challenge. Initial research efforts in the area of low-level implementations of graph algorithms and GPU graph

[1] NVIDIA's TESLA P100 GPU based on its latest Pascal architecture achieves 10.6Tflops of its peak single-precision floating point performance and up to 732 Gbytes/sec memory bandwidth.

processing systems is promising [18, 21, 34, 43, 59]. Unfortunately, parallel graph analytics on the GPU still faces two major challenges: (i) low-level implementations of specific graph primitives are difficult to develop even by the most skilled programmers, and do not generalize well to a variety of graph primitives since the implementations often couples a specific graph computation to a specific type of parallel graph operation; (ii) high-level GPU programming models for graph analytics often recapitulate CPU programming models and do not compare favorably in performance with specialized implementations due to different kinds of overhead introduced by maintaining a high-level framework.

1.1 Dissertation Contributions

This dissertation seeks to resolve the conflict of programmability and high -performance for graph analytics on the GPU by designing a GPU-specific data -centric programming model and building a graph analytics system on the GPU that not only allows quick prototyping of new graph primitives but also delivers the performance of customized, complex GPU hardwired graph primitives. We show that with appropriate high-level programming model and low-level optimizations, parallel graph analytics on the GPU can be simple and efficient. The code developed as part of this thesis is publicly available, and has already been used by various researchers for benchmarking and developing their own GPU graph analytics solutions.

This thesis makes the following contributions to the field of graph analytics on GPUs:

Data-centric abstraction: We present a novel data-centric abstraction for graph operations that allows programmers to develop graph primitives at a high level of abstraction while simultaneously delivering high performance. This abstraction, unlike the abstractions of previous GPU programmable frameworks, is able to elegantly incorporate several profitable optimizations, which previously were only applied to different individual graph algorithm implementations on the GPU, into the core of its implementation. These optimizations include kernel fusion, push-pull traversal, idempotent traversal, priority queues, and various workload mapping strategies.

Graph analytics system: We design and implement a new graph analytics system, Gunrock [88], which contains a set of simple and flexible graph operation APIs that can express a wide range of graph primitives at a high level of abstraction. We also show how the combination and tuning of several GPU-specific optimization strategies for memory efficiency, load balancing, and workload management can achieve high performance. All of our graph primitives achieve comparable performance to their hardwired counterparts and significantly outperform previous programmable GPU abstractions.

Graph primitives: With Gunrock's expressiveness, we implement a large set of graph primitives. The types of algorithms span from those can be presented using existing GPU graph analytics system such as traversal-based algorithms and ranking algorithms, to those cannot fit existing GPU graph programming model such as triangle counting, bipartite-graph-based algorithms and label propagation/clustering algorithm.

1.2 Dissertation Outline

This chapter will continue with a description of preliminaries of graph analytics and GPU architecture. Chapter 2 will provide a background survey of current parallel graph analytics systems and programming models and then describes our data-centric abstraction, shows its difference with other programming models and thus provides why this programing model fits best for the GPU architecture. Chapter 3 will describe various low-level optimizations that we have integrated into our graph analytics system, Gunrock, and show how to combine and tune them to get the best performance for different graph types and algorithms. Chapter 4 will describe the design of Gunrock system, and also show the programmability of our data-centric programming model by providing a lightweight graph analytics system on the GPU which achieves comparable performance with Gunrock with a minimum code size. Chapter 5 presents the expressiveness of our data-centric programming model by listing a rich set of graph primitives that could be presented using simple graph operations we define in Gunrock. Chapter 6 gives detailed performance characterization which identifies several factors that are critical to high-performance GPU graph analytics. Finally, we describe open problems and conclude in

Chapter 7.

1.3 Preliminaries and Notation

1.3.1 Graphs

A graph is an ordered pair $G = (V, E, w_e, w_v)$ comprised of a set of nodes V together with a set of edges E, where $E \subseteq V \times V$. For weighted graph, w_e and w_v are two weight functions that show the weight values attached to edges and nodes in the graph. A graph is undirected if for all $v, u \in V : (v, u) \in E \iff (u, v) \in E$. Otherwise, it is directed. For directed graph, $N^+(v)$ and $d^+(v)$ denote the out-going neighbors and out-degree of node v in the graph, $N^-(v)$ and $d^-(v)$ denote the in-coming neighbors and in-degree of node v in the graph. For undirected graph, $N(v)$ and $d(v)$ denote the neighbors and degree of node v in the graph. In this thesis, we refer to graph analytics as the algorithms running on graphs trying to determine strength and direction of relationships between objects in graphs. In graph analytics, a **vertex frontier** represents a subset of vertices $U \subseteq V$, an **edge frontier** represents a subset of edges $I \subseteq E$, and the **adjacency list** format in this thesis stores ordered node IDs for each node's neighbors in an neighbor list array and the offset indices to the neighbor list array in an offset array.

1.3.2 GPU Architecture

Modern NVIDIA GPUs are throughput-oriented many-core processors that use massive parallelism to get very high peak computational throughput and hide memory latency. Kepler-based GPUs can have up to 15 vector processors, termed streaming multiprocessors (SMX), each containing 192 parallel processing cores, termed streaming processors (SP). NVIDIA GPUs use the Single Instruction Multiple Thread (SIMT) programming model to achieve data parallelism. GPU programs called *kernels* run on a large number of parallel threads. Each set of 32 threads forms a divergent-free group called a *warp* to execute in lockstep in a Single Instruction Multiple Data (SIMD) fashion. These warps are then grouped into cooperative thread arrays (CTA) called *blocks* whose threads can communicate through a pool of on-chip shared memory. All SMXs

share an off-chip global DRAM.

1.3.3 Parallel Primitives

We will build the basic operators in our graph analytics system upon basic parallel primitives, prefix sum (scan), reduce, compact, and merge/intersection. **Prefix sum (scan) [47]** takes a binary associative operator $\oplus$ with identity I, and a sequence of n elements, $[a_0, a_1, ..., a_{n-1}]$, and returns the sequence $[I, (I \oplus a_0), (I \oplus a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$ if it is exclusive and $[(I \oplus a_0), (I \oplus a_0 \oplus a_1), ..., (I \oplus a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$ if it is inclusive. **Reduce [9]** takes the same arguments as prefix sum, but only returns the resulting sum $I \oplus a_0 \oplus a_1 \oplus ... \oplus a_{n-1}$. **Compact [8]** applies a selection criterion f to selectively copy items from a specified input sequence A to a compact output sequence. **Merge** takes two sorted sequences A and B respectively, and returns a sequence containing all the elements in A and B in sorted order. **Intersection** can be implemented with minor modification of merge. All these parallel primitives have their corresponding GPU implementations [8, 27, 35, 74]. In this thesis we use the-state-of-the-art GPU implementations of these parallel primitives from moderngpu [6] and CUB [58].

1.3.4 Atomic Operations

In the context of GPU computing, an atomic operation is an uninterruptable read-modify-write memory operation that requested by threads and updates a value at a specific address. Using atomic operations in GPU threads can help implement a wide range of generic data structure and algorithms, but will also serializes contentious updates from multiple threads. In this thesis, we use three atomic operations:**atomicCAS**,**atomicAdd**, and **atomicMin**. **atomicCAS** takes a memory location *addr*, and old value *oldV* and a new value *newV*, modifies the value stored at *addr* to *newV* and returns *true* if the value is equal to *oldV*, or returns *false* otherwise. **atomicAdd** and **atomicMin** reads a word at some address in global or shared memory, adds a number to it, or compute min value with a number, and writes the result back to the same address. The operations are atomic in the sense that they are guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is

complete [62].

1.3.5 Parallel Models for Graph Analytics

Bulk Synchronous Parallel (BSP) is a popular parallel programming models that systems built on GPUs often follow.

A BSP model consists of: 1) components capable of processing and/or local memory transactions; 2) a network that routes message between pairs of such components, and 3) a hardware facility that allows for the synchronization of all or a subset of components [85]. A BSP algorithm relies heavily on a computation proceeds in a series of global supersteps, which consists of three components:

Concurrent computation: Every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication.

Communication: The processes exchange data between themselves to facilitate remote data storage capabilities.

Barrier synchronization: When a process reaches the barrier, it waits until all other processes have reached the same barrier.

BSP operations are well-suited for efficient implementation on the GPU because they exhibit enough parallelism to keep the GPU busy and do not require expensive fine-grained synchronization or locking operations.

Gather-Apply-Scatter (GAS) is a parallel abstraction designed specifically for graph analytics. It contains three phases: gather, apply, and scatter.

Gather collects information from the neighbors of a node using a generalized binary operator to perform a parallel reduction over the neighborhood.

Apply integrates the information from the Gather phase, updating the state of the node.

Scatter redistributes information to the neighbors of a node.

1.3.6 Graph Primitives

This section lists various graph primitives we study in this thesis.

Breadth-First Search (BFS): For a graph G and source node s, returns a sequence containing number of hops from source node s for every node reachable from s in G.

Single-Source Shortest Path (SSSP): For a weighted graph $G = (V, E, w)$ and a source node s, computes the shortest path distance from s to each node in V and returns the shortest paths from s to all other reachable nodes in G.

Betweenness Centrality (BC): For a weighted graph $G = (V, E, w)$, computes a centrality value for each node or each edge in G and returns a sequence of node IDs in sorted order of their centrality values. In this thesis, without losing generality, we simplify the algorithm to have a non-weighted graph $G = (V, E)$ as input.

Connected Component Labeling (CC): For an undirected graph G returns a labeling L sorted in the order of number of nodes in the connected component, such that for two nodes u and v, $L(u) = L(v)$ if u and v belongs in the same connected component, and $L(u) \neq L(v)$ otherwise.

Pagerank (PR): For a graph G returns a sequence containing ranking scores for every node in G represents their relative importance.

Triangle Counting (TC) and clustering coefficient: For an undirected graph G, TC returns a sequence containing number of triangles each edge joins and the total number of triangles in G. Triadic closure, a concept in social network theory, is the property among three nodes A, B, and C, such that if a strong tie exists between A-B and A-C, there is a weak or strong tie between B-C. Clustering coefficient, as a measure for the presence of triadic

closure, can be computed as a byproduct of TC.

Bipartite Graph Ranking: For a bipartite graph $G = (U, V, E)$ where U and V are two disjoint sets connected by E, bipartite graph ranking algorithm such as Hyperlink-Induced Topic Search (HITS), Stochastic Approach for Link-Structure Analysis (SALSA), and Twitter's Who-to-Follow (WTF) all return a sequence containing ranking scores for every node in G.

1.3.7 Experiment Environment

We ran all experiments in this thesis on a Linux workstation with 2×3.50 GHz Intel 4-core, hyperthreaded E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. GPU programs were compiled with NVIDIA's nvcc compiler (version 8.0.44) with the -O3 flag. All results ignore transfer time (both disk-to-memory and CPU-to-GPU).

Chapter 2

Data-Centric Abstraction for Graph Analytics on the GPU

2.1 Large-Scale Graph Analytics Frameworks

This section discusses the research landscape of large-scale graph analytics frameworks in four fields:

1. Single-node CPU-based systems, which are in common use for graph analytics today, but whose serial or coarse-grained-parallel programming models are poorly suited for a massively parallel processor like the GPU;

2. Distributed CPU-based systems, which offer scalability advantages over single-node systems but incur substantial communication cost, and whose programming models are also poorly suited to GPUs;

3. GPU "hardwired," low-level implementations of specific graph primitives, which provide a proof of concept that GPU-based graph analytics can deliver best-in-class performance. However, best-of-class hardwired primitives are challenging to even the most skilled programmers, and their implementations do not generalize well to a variety of graph primitives; and

4. High-level GPU programming models for graph analytics, which often recapitulate CPU programming models. The best of these systems incorporate generalized load-balance strategies and optimized GPU primitives, but they generally do not compare favorably in performance with hardwired primitives due to the overheads inherent in a high-level framework and the lack of primitive-specific optimizations.

Section 2.1.1 gives a survey of CPU graph analytics systems. Section 2.1.2 and section 2.1.3 discuss specialized parallel graph algorithms on CPUs and GPUs as well as graph analytics systems on the GPU.

2.1.1 Single-node and Distributed CPU-based Systems

Parallel graph analytics frameworks provide high-level, programmable, high-performance abstractions. The Boost Graph Library (BGL) is among the first efforts towards this goal, though its serial formulation and C++ focus together make it poorly suited for a massively parallel architecture like a GPU. Designed using the generic programming paradigm, the parallel BGL [30] separates the implementation of parallel algorithms from the underlying data structures and communication mechanisms. While many BGL implementations are specialized per algorithm, its breadth_first_visit pattern (for instance) allows sharing common operators between different graph algorithms.

There are two major framework families for CPU-based large-scale graph processing system: Pregel and GraphLab.

Pregel [53] is a Google-initiated programming model and implementation for large-scale graph computing that follows the BSP model. A typical application in Pregel is an iterative convergent process consisting of global synchronization barriers called super-steps. The computation in Pregel is vertex-centric and based on message passing. Its programming model is good for scalability and fault tolerance. However, standard graph algorithms in most Pregel-like graph processing systems suffer slow convergence on large-diameter graphs and load imbalance on scale-free graphs. Apache Giraph is an open source implementation of Google's Pregel. It is a

popular graph computation engine in the Hadoop ecosystem initially open-sourced by Yahoo!. GraphLab [52] is an open-source large scale graph processing library that relies on the shared memory abstraction and the gather-apply-scatter (GAS) programming model. It allows asynchronous computation and dynamic asynchronous scheduling. By eliminating message-passing, its programming model isolates the user-defined algorithm from the movement of data, and therefore is more consistently expressive. PowerGraph [25] is an improved version of GraphLab for power-law graphs. It supports both BSP and asynchronous execution. For the load imbalance problem, it uses vertex-cut to split high-degree vertices into equal degree-sized redundant vertices. This exposes greater parallelism in natural graphs. GraphChi [46] is a centralized system that can process massive graphs from secondary storage in a single machine. It introduces a novel graph partitioning method called Parallel Sliding Windows (PSW), which sorts the edges by their source node ID to provide load balancing.

Beyond these two families, there are several other notable graph processing libraries such as GraphX and Help; shared-memory-based systems such as Ligra and Galois; and domain-specific languages such as Green-Marl.

GraphX [26] is a distributed graph computation framework that unifies graph-parallel and data-parallel computation. It provides a small, core set of graph-parallel operators expressive enough to implement the Pregel and PowerGraph abstractions, yet is simple enough to be cast in relational algebra. Help is a library that provides high-level primitives for large-scale graph processing [70]. Using the primitives in Help is more intuitive and faster than using the APIs of existing distributed systems. Ligra [78] is a CPU-based graph processing framework for shared memory. It uses a similar operator abstraction for doing graph traversal. Its lightweight implementation is targeted at shared memory architectures and uses CilkPlus for its multithreading implementation. Galois [61, 68] is a graph system for shared memory based on a different operator abstraction that supports priority scheduling and dynamic graphs and processes on subsets of vertices called active elements. However, their model does not abstract the internal details of the loop from the user. Users have to generate the active elements set directly for different graph algorithms. Green-Marl [38] is a domain-specific language for writing graph analysis algorithms on shared

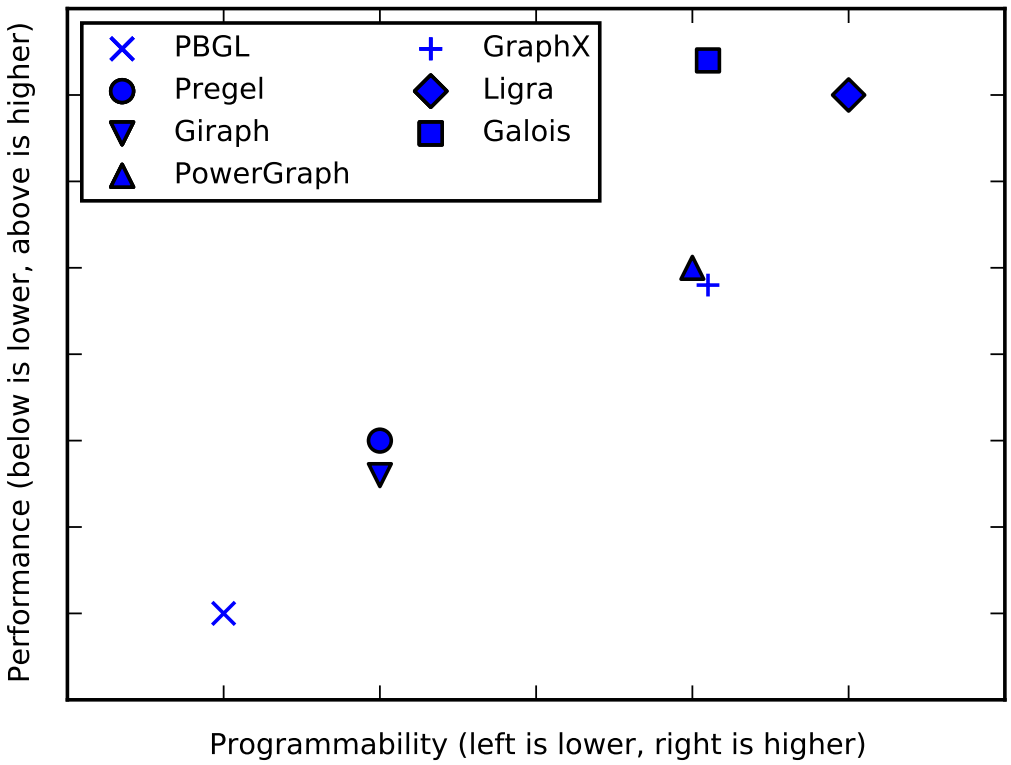


Figure 2.1: *Qualitative performance-vs-programmability scatter plot of various CPU graph analytics systems.*

memory with built-in breadth-first search (BFS) and depth-first search (DFS) primitives in its compiler. Its language approach provides graph-specific optimizations and hides complexity. However, the language does not support operations on arbitrary sets of vertices for each iteration, which makes it difficult to use for traversal algorithms that cannot be expressed using a BFS or DFS.

Figure 2.1 shows the qualitative performance-vs -programmability scatter plot of various CPU graph analytics systems we have discussed above. We exclude Help and Green-Marl since their performance cannot be easily evaluated from their papers.

2.1.2 Specialized Parallel Graph Algorithms

Recent work has developed numerous best-of-breed, hardwired implementations of many graph primitives.

Breadth-first search is among the first few graph primitives researchers developed on the GPU due to its representative workload pattern and its fundamental role as the building block primitive to several other traversal-based graph primitives. Harish et al. [34] first proposed a quadratic GPU BFS implementation that maps each vertex's neighbor list to one thread. Hong et al. [37] improved on this algorithm by mapping workloads to a series of virtual warps and letting an entire warp to cooperatively strip-mine the corresponding neighbor list. Merrill et al. [59]'s linear parallelization of the BFS algorithm on the GPU had significant influence in the field. They proposed an adaptive strategy for load-balancing parallel work by expanding one node's neighbor list to one thread, one warp, or a whole block of threads. With this strategy and a memory-access efficient data representation, their implementation achieves high throughput on large scale-free graphs. Beamer et al.'s recent work on a very fast BFS for shared memory machines [7] uses a hybrid BFS that switches between top-down and bottom-up neighbor-list-visiting algorithms according to the size of the frontier to save redundant edge visits. Enterprise [50], a GPU-based BFS system, introduces a very efficient implementation that combines the benefits of direction optimization, Merrill et al.'s adaptive load-balancing workload mapping strategy, and a status-check array. BFS-4K [14] is a GPU BFS system that improves the virtual warp method to a per-iteration dynamic one and uses dynamic parallelism for better load balancing.

Connected components can be implemented as a BFS-based primitive. The current fastest connected-component algorithm on the GPU is Soman et al.'s work [81] based on a PRAM connected-component algorithm [31] that initializes each vertex in its own component, and merges component IDs by traversing in the graph until no component ID changes for any vertex.

There are several parallel betweenness centrality implementations on the GPU based on the work from Brandes [10]. Pande and Bader [66] proposed the first BC implementation on the GPU using a quadratic BFS . Sariyüce et al. [72] adopted PowerGraph's vertex-cut method

(which they termed vertex virtualization) to improve the load balancing and proposed a stride-CSR representation to reorganize adjacency lists for better memory coalescing. McLaughlin and Bader [57] developed a work-efficient betweenness centrality algorithm on the GPU that combines a queue-based linear multi-source BFS-based work-efficient BC implementation with an edge-parallel BC implementation, It dynamically chooses between the two according to frontier size and scales linearly on up to 192 GPUs.

Davidson et al. [18] proposed a delta-stepping-based [60] work-efficient single-source shortest path algorithm on the GPU that explores a variety of parallel load-balanced graph traversal and work organization strategies to outperform previous parallel methods that are either based on quadratic BFS [34] or the Bellman-Ford algorithm (LonestarGPU 2.0 [13]).

After we discuss the Gunrock abstraction in Section 2.2, we will discuss how to map these specialized graph algorithms to Gunrock and the differences in Gunrock's implementations.

2.1.3 High-level GPU Programming Models

Several works target the construction of a high-level GPU graph processing library that delivers both good performance and good programmability. We categorize these into two groups by programming model: (1) BSP/Pregel's message-passing framework and (2) the GAS model.

In Medusa [92], Zhong and He presented their pioneering work on parallel graph processing using a message-passing model called Edge-Message-Vertex (EMV). It is the first high-level GPU graph analytics system that can automatically executes different user-defined APIs for different graph primitives, which increases the programmability to some extent. However, its five types of user-defined API—namely, *ELIST, EDGE, MLIST, MESSAGE, VERTEX*—are still vertex-centric and need to be split into several source files. Moreover, Medusa does not have a fine-grained load balancing strategy for unevenly distributed neighbor lists during graph traversal, which makes its performance on scale-free graphs incomparable to specialized graph primitives.

Two more recent works that follow message-passing approach both improves the execution model. Totem [23] is a graph processing engine for GPU-CPU hybrid systems. It either processes

the workload on the CPU or transmits it to the GPU according to a performance estimation model. Its execution model can potentially solve the long-tail problem (where the graph has a large diameter with a very small amount of neighbors to visit per iteration) on GPUs, and overcome GPU memory size limitations. Totem's programming model allows users to define two functions—*algo_compute_func* and *msg_combine_func*— to apply to various graph primitives. However, because its API only allows direct neighbor access, it has limitations in algorithm generality. Frog [76] is a lock-free asynchronous parallel graph processing framework with a graph coloring model. It has a preprocessing step to color the graph into sets of conflict-free vertices. Although vertices with the same color can be updated lock-free in a parallel stage called color-step, on a higher level the program still needs to process color-steps in a BSP style. Within each color-step, their streaming execution engine is message-passing-based with a similar API to Medusa. The preprocessing step of the color model does accelerate the iterative convergence and enables Frog to process graphs that do not fit in a single GPU's memory. However, the CPU-based coloring algorithm is inefficient, also due to the limitation of its programming model, as performance is restricted by visiting all edges in each single iteration.

The GAS abstraction was first applied on distributed systems [25]. PowerGraph's vertex-cut splits large neighbor lists, duplicates node information, and deploys each partial neighbor list to different machines. Working as a load balancing strategy, it replaces the large synchronization cost in edge-cut into a single-node synchronization cost. This is a productive strategy for multi-node implementations. GAS offers the twin benefits of simplicity and familiarity, given its popularity in the CPU world. VertexAPI2 [20] is the first GPU high-level graph analytics system that strictly follows the GAS model. It defines two graph operators: gatherApply, the combination of GAS's gather step and apply step, and scatterActivate, GAS's scatter step. Users build different graph primitives by defining different functors for these operators. VertexAPI2 has four types of functors:*gatherReduce*, *gatherMap*,*scatter*, and *apply*. Underneath their abstraction, VertexAPI2 uses state-of-the-art GPU data primitives based on two-phase decomposition [6]. It shows both better performance and better programmability compared to message-passing-based GPU libraries. MapGraph [21] replicates VertexAPI2's framework and managed to integrate both moderngpu's load-balanced search [6] and Merrill et al.'s [59] dynamic grouping workload

Metrics	Medusa	Totem	Frog	VertexAPI2	MapGraph	CuSha	Gunrock
programming model	m-p	m-p	m-p	GAS	GAS	GAS	data-centric
operator flexibility	v-c, e-c	v-c	v-c	v-c	v-c	v-c	v-c, e-c
load balancing	no	limited	limited	fine-grained	fine-grained	fine-grained	fine-grained
preprocessing	no	no	coloring	no	no	G-shard	no
execution model	BSP	BSP,hybrid	semi-async	BSP	BSP	BSP	BSP

Table 2.1: *Detailed comparison of different high-level GPU graph analytics systems. m-p means message-passing based model, v-c and e-c mean vertex-centric and edge-centric respectively. Note that part of load balancing work in Frog and CuSha are done offline during the coloring model and G-shard generation process respectively.*

mapping strategy to increase its performance. CuSha [43] is also a GAS model-based GPU graph analytics system. It solves the load imbalance and GPU underutilization problem with a GPU adoption of GraphChi's PSW. They call this processing step "G-Shard" and combine it with a concatenated window method to group edges from the same source IDs.

Table 2.1 compares Gunrock with the above high-level GPU-based graph analytics systems on various metrics. With our novel data-centric programming model and its efficient implementation, Gunrock is the only high-level GPU-based graph analytics system that supports both vertex-centric and edge-centric operations, with runtime fine-grained load balancing strategies without requiring any preprocessing of input datasets.

The goal of this thesis is to design and implement a GPU graph analytics system that finds its location in the upper right corner of Figure 2.1. To achieve this goal, we need efforts to improve both programmability and performance of GPU graph analytics.

From a programmability perspective, Gunrock aims to not only define a programming model that can abstract most common operations in graph analytics at a high level while being flexible enough to be used to express various types of graph analytics tasks, but also matches this programming model to the GPU's high throughput, data-centric, massively parallel execution model so that the generality of the model does not hurt its performance.

From a performance perspective, Gunrock attempts to build its low-level optimizations on top

of the state-of-the-art basic data parallel primitives on the GPU, designing them in a way so that we can use different combinations and parameter-tuning methods in our graph operations to achieve comparable performance to specialized GPU graph algorithm implementations.

The following part of this chapter focuses on the design of our graph analytics programming model on the GPU, its motivation, design, key differences from other CPU and GPU graph analytics systems, and the benefits it brings to achieving our goal of creating a GPU graph analytics system with both high programmability and high performance.

2.2 Data-Centric Abstraction

A common finding from most CPU and GPU graph analytics systems is that most graph analytics tasks can be expressed as iterative convergent processes. By "iterative," we mean operations that may require running a series of steps repeatedly; by "convergent," we mean that these iterations allow us to approach the correct answer and terminate when that answer is reached. Mathematically, iterative convergent process is a series of operations $f_0, f_1, ..., f_{n-1}$ that operate on graph data G, where $G_i = f(G_{i-1})$, until at iteration i, the stop condition function $C(G_i, i)$ returns true.

Both Pregel and PowerGraph focus on sequencing steps of *computation*. Where Gunrock differs from these two frameworks, and other GPU-based frameworks based on them, is in our abstraction. Rather than focusing on sequencing steps of *computation*, we instead focus on manipulating a data structure, the *frontier* of vertices or edges that represents the subset of the graph that is actively participating in the computation. Gunrock's data-centric framework design not only covers the features of other frameworks, but also provides high performance and is flexible enough to be expanded with new graph operators as long as they operate on multiple input frontiers with graph data and generate multiple output frontiers. Because of this design, we claim that thinking about graph processing in terms of manipulations of frontier data structures is the right abstraction for the GPU. We support this statement qualitatively in this chapter and quantitatively in chapter 6.

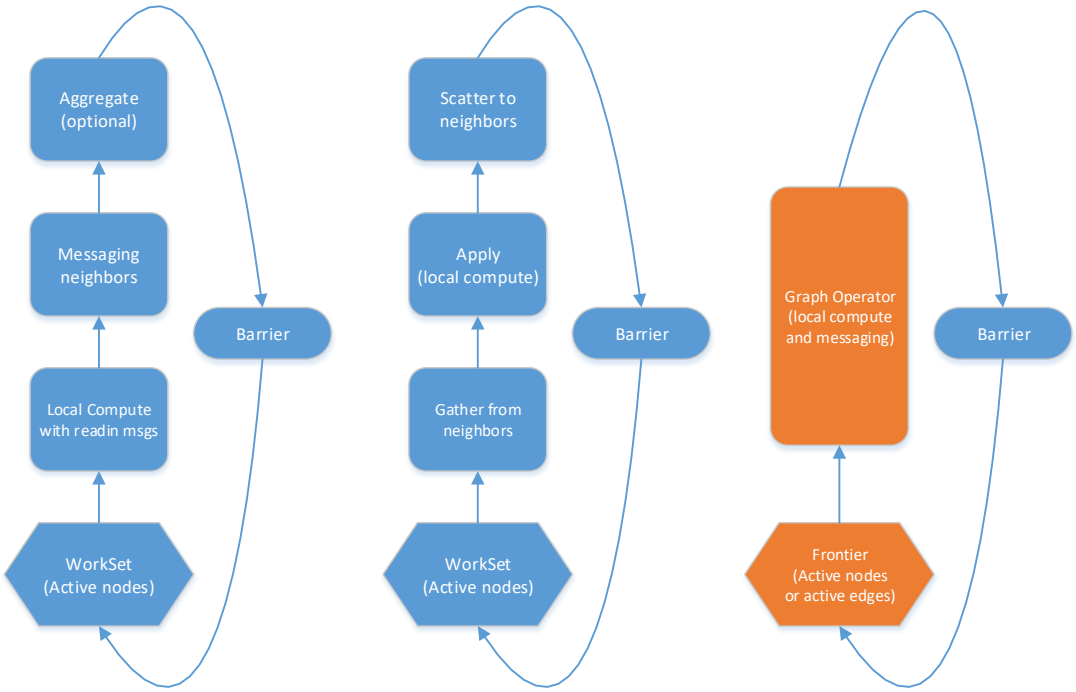


Figure 2.2: *Iterative convergence process presented using Pregel's message passing model, PowerGraph's GAS model, and Gunrock's data-centric model.*

One important consequence of designing our abstraction with a data-centered focus is that Gunrock, from its very beginning, has supported both node and edge frontiers, and can easily switch between them within the same graph primitive. We can, for instance, generate a new frontier of neighboring edges from an existing frontier of vertices. In contrast, gather-apply-scatter (PowerGraph) and message-passing (Pregel) abstractions are focused on operations on vertices and either cannot support edge-centric operations or could only support them with heavy redundancy within their abstractions.

In our abstraction, we expose bulk-synchronous "steps" that manipulate the frontier, and programmers build graph primitives from a sequence of steps. Different steps may have dependencies between them, but individual operations within a step can be processed in parallel. For instance, a computation on each vertex within the frontier can be parallelized across vertices, and updating the frontier by identifying all the vertices neighboring the current frontier can also be parallelized across vertices.

The graph primitives we describe in this dissertation use four irregular traversal operators: ad-

vance, filter, neighborhood reduction, and segmented intersection, and one compute operator, which is often fused with one of the traversal operators (Figure 2.3). Each graph operator manipulates the frontier in a different way. The input frontier of each operator contains either node IDs or edge IDs that specify on which part of the graph we are going to perform our computations. The traversal operators would traverse the graph and generate output frontier. Each input item can map to zero, one, or more output items according to different traversal operators. According to different implementation methods, a single thread can process one or more input items and generate one or more output items. We provide the operator description here and discuss the implementation using high-performance GPU data-parallel primitives in Chapter 4.

Advance An *advance* operator generates a new frontier from the current frontier by visiting the neighbors of the current frontier. Each input item maps to multiple output items which are in that input item's neighbor list. A frontier can consist of either vertices or edges, and an advance step can input and output either kind of frontier. Advance is an irregularly-parallel operation for two reasons: (1) different vertices in a graph have different numbers of neighbors and (2) vertices share neighbors, so an efficient advance is the most significant challenge of a GPU implementation.

The generality of Gunrock's advance allows us to use the same advance implementation across a wide variety of interesting graph operations. According to the type of input frontier and output frontier, Gunrock supports 4 kinds of advance: V-to-V, V-to-E, E-to-V, and E-to-E, where E represents edge frontier, and V represents vertex frontier. For instance, we can utilize Gunrock advance operators to: 1) visit each element in the current frontier while updating local values and/or accumulating global values (e.g., BFS distance updates); 2) visit the node or edge neighbors of all the elements in the current frontier while updating source vertex, destination vertex, and/or edge values (e.g., distance updates in SSSP); 3) generate edge frontiers from vertex frontiers or vice versa (e.g., BFS, SSSP, SALSA, etc.); or 4) pull values from all vertices 2 hops away by starting from an edge frontier, visiting all the neighbor edges, and returning the far-end vertices of these neighbor edges.

Filter A *filter* operator generates a new frontier from the current frontier by choosing a subset of the current frontier based on programmer-specified criteria. Each input item maps to zero or one output item. Though filtering is an irregular operation, using parallel scan for efficient filtering is well-understood on GPUs. Gunrock's filters can either 1) split vertices or edges based on a filter (e.g., SSSP's delta-stepping), or 2) compact out filtered items to throw them away (e.g., duplicate vertices in BFS and edges with both end nodes belong to the same component in CC).

Neighborhood Reduction An *neighborhood reduction* operator, similar to advance operator, generates a new frontier from the current frontier by visiting the neighbors of the current frontier. As advance, a neighborhood reduction operator can input and output either kind of frontier. The difference between neighborhood reduction and advance is that neighborhood reduction will operate a user-specified parallel reduction while doing the neighbor expanding. Neighborhood reduction is also an irregularly-parallel operation for the same reason as advance. Neighborhood reduction is the key operator behind several graph primitives which are difficult to implement using other GPU frameworks, such as MIS, graph coloring, and LP.

Segmented Intersection A *segmented intersection* operator takes two input frontiers with the same length, or an edge frontier. We call input items with the same index in two input frontiers a pair. If the input is an edge frontier, we treat each edge's two nodes as an input item pair.One input item pair maps to multiple output items, which are the intersection of the neighbor lists of two input items. Segmented intersection is the key operator in TC . The byproducts of this operator are global triangle counting and clustering coefficient. The output frontier could be combined with the two input frontiers to enumerate all the triangles in the graph.

Compute A programmer-specified *compute* operator is used in all four traversal operators. It defines an operation on all elements (vertices or edges) in the current frontier; Gunrock then performs that operation in parallel across all elements and the order does not matter. The computaion can access and modify global memory through a device pointer of a structure

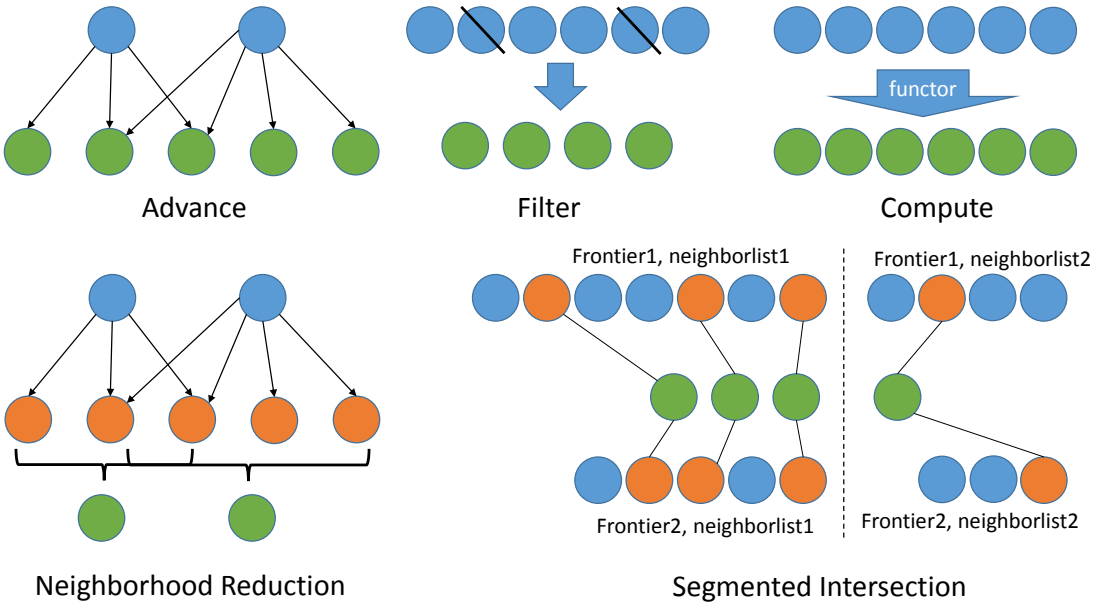


Figure 2.3: *Five operators in Gunrock's data-centric abstraction convert a current frontier (in blue) into a new frontier (in green).*

of array that stores per-node and/or per-edge data. Because this parallelism is regular, computation is straightforward to parallelize in a GPU implementation. Many simple graph primitives (e.g., computing the degree distribution of a graph) can be expressed as a single computation operator.

Gunrock primitives are assembled from a sequence of these five operators, which are executed sequentially: one step completes all of its operations before the next step begins. Typically, Gunrock graph primitives run to convergence, which on Gunrock usually equates to an empty frontier; as individual elements in the current frontier reach convergence, they can be filtered out of the frontier. Programmers can also use other convergence criteria such as a maximum number of iterations or volatile flag values that can be set in a computation step.

Example: Expressing SSSP in programmable GPU frameworks Now we use an example to show how different programmable CPU/GPU frameworks express a graph primitive to further study the key difference between Gunrock's data-centric abstraction and other frameworks. We choose SSSP because it is a reasonably complex graph primitive that computes the shortest path from a single node in a graph to every other node in the graph. We assume weights

between nodes are all non-negative, which permits the use of Dijkstra's algorithm and its parallel variants. Efficiently implementing SSSP continues to be an interesting problem in the GPU world [12, 18, 19].

The iteration starts with an input frontier of active vertices (or a single vertex) initialized to a distance of zero. First, SSSP enumerates the sizes of the frontier's neighbor list of edges and computes the length of the output frontier. Because the neighbor edges are unequally distributed among the frontier's vertices, SSSP next redistributes the workload across parallel threads. This can be expressed within an advance operator. In the final step of the advance operator, each edge adds its weight to the distance value at its source value and, if appropriate, updates the distance value of its destination vertex. Finally, SSSP removes redundant vertex IDs (specific filter), decides which updated vertices are valid in the new frontier, and computes the new frontier for the next iteration.

Algorithm 1 provides more detail of how this algorithm maps to Gunrock's abstraction.

Algorithm 1 Single-Source Shortest Path, expressed in Gunrock's abstraction

```
1: procedure Set_Problem_Data(G, P, root)
2:     P.labels[1..G.verts] ← ∞
3:     P.preds[1..G.verts] ← −1
4:     P.labels[root] ← 0
5:     P.preds[root] ← src
6:     P.frontier.Insert(root)
7: end procedure
8:
9: procedure UpdateLabel(s_id, d_id, e_id, P)
10:    new_label ← P.labels[s_id] + P.weights[e_id]
11:    return new_label < atomicMin(P.labels[d_id], new_label)
12: end procedure
13:
14: procedure SetPred(s_id, d_id, P)
15:    P.preds[d_id] ← s_id
16:    P.output_queue_ids[d_id] ← output_queue_id
17: end procedure
18:
19: procedure RemoveRedundant(node_id, P)
20:    P.priority_level[node_id] ← ComputePriorityLevel(P.labels[s_id])
21:    pick_node ← P.priority_level[node_id] == P.cur_priority_level
22:    not_redundant ← P.output_queue_id[node_id] == output_queue_id
23:    return not_redundant&&pick_node
24: end procedure
25:
26: procedure SSSP_Enactor(G, P, root)
27:    Set_Problem_Data(G, P, root)
28:    while P.frontier.Size() > 0 do
29:        Advance(G, P, UpdateLabel, SetPred)
30:        Filter(G, P, RemoveRedundant)
31:    end while
32: end procedure
```

Gunrock maps one SSSP iteration onto three Gunrock operators: (1) *advance*, which computes the list of edges connected to the current vertex frontier and (transparently) load-balances their execution; (2) *compute*, to update neighboring vertices with new distances; and (3) *filter*, to generate the final output frontier by removing redundant nodes, optionally using a 2-level priority queue, whose use enables delta-stepping (a binning strategy to reduce overall workload [18, 60]). With this mapping in place, the traversal and computation of path distances is simple and intuitively described, and Gunrock is able to create an efficient implementation that fully utilizes

the GPU's computing resources in a load-balanced way.

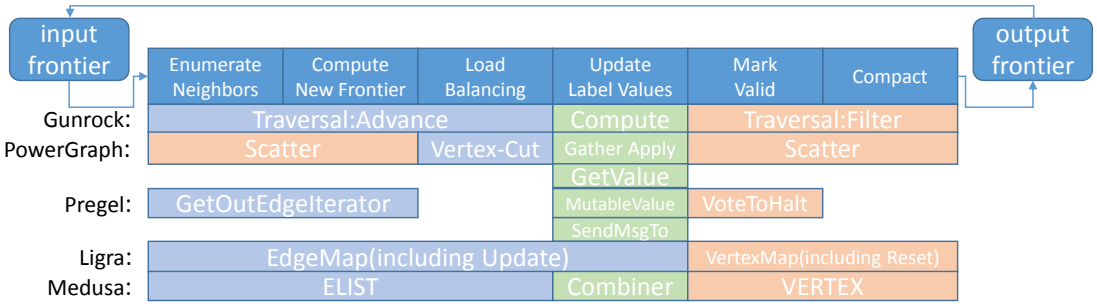


Figure 2.4: *Operations that make up one iteration of SSSP and their mapping to Gunrock, PowerGraph (GAS) [25], Pregel [53], Ligra [78], and Medusa [92] abstractions.*

2.2.1 Alternative Abstractions

In this section we discuss Gunrock's design choices compared to several alternative abstractions designed for graph processing on various architectures.

Gather-apply-scatter (GAS) abstraction Recently, Wu et al. compared Gunrock vs. two GPU GAS frameworks, VertexAPI2 and MapGraph [90], demonstrating that Gunrock had appreciable performance advantages over the other two frameworks. One of the principal performance differences they identified comes from the significant fragmentation of GAS programs across many kernels that we discuss in more detail in chapter 3. Applying automatic kernel fusion to GAS+GPU implementations could potentially help close their performance gap [64].

At a more fundamental level, we found that a compute-focused programming model like GAS was not flexible enough to manipulate the core frontier data structures in a way that enabled powerful features and optimizations such as push-pull and two-level priority queues; both fit naturally into Gunrock's abstraction. We believe bulk-synchronous operations on frontiers are a better fit than GAS for forward-looking GPU graph programming frameworks.

Message-passing Pregel [53] is a vertex-centric programming model that only provides data parallelism on vertices. For graphs with significant variance in vertex degree (e.g., power-

law graphs), this would cause severe load imbalance on GPUs. The traversal operator in Pregel is general enough to apply to a wide range of graph primitives, but its vertex-centric design only achieves good parallelism when nodes in the graph have small and evenly-distributed neighborhoods. For real-world graphs that often have uneven distribution of node degrees, Pregel suffers from severe load imbalance. The Medusa authors note the complexity of managing the storage and buffering of these messages, and the difficulty of load-balancing when using segmented reduction for per-edge computation. Though they address both of these challenges in their work, the overhead of *any* management of messages is a significant contributor to runtime. Gunrock prefers the less costly direct communication between primitives and supports both push-based (scatter) communication and pull-based (gather) communication during traversal steps.

CPU strategies Ligra's powerful load-balancing strategy is based on CilkPlus, a fine-grained task-parallel library for CPUs. Despite promising GPU research efforts on task parallelism [15, 84], no such equivalent is available on GPUs, thus we implement our own load-balancing strategies within Gunrock. Galois, like Gunrock, cleanly separates data structures from computation; their key abstractions are ordered and unordered set iterators that can add elements to sets during execution (such a dynamic data structure is a significant research challenge on GPUs). Galois also benefits from speculative parallel execution whose GPU implementation would also present a significant challenge. Both Ligra and Galois scale well within a node through inter-CPU shared memory; inter-GPU scalability, both due to higher latency and a lack of hardware support, is a much more manual, complex process.

Help's Primitives Help [70] characterizes graph primitives as a set of functions that enable special optimizations for different primitives at the cost of losing generality. Its Filter, Local Update of Vertices (LUV), Update Vertices Using One Other Vertex (UVUOV), and Aggregate Global Value (AGV) are all Gunrock filter operations with different computations. Aggregating Neighbor Values (ANV) maps to the advance operator in Gunrock. We also successfully implemented Form Supervertices (FS) in Gunrock using two filter

passes, one advance pass, and several other GPU computing primitives (sort, reduce, and scan).

Asynchronous execution Many CPU and GPU frameworks (e.g., Galois, GraphLab, and Frog) efficiently incorporate asynchronous execution, but the GPU's expensive synchronization or locking operations would make this a poor choice for Gunrock. We do recover some of the benefits of prioritizing execution through our two-level priority queue.

```
// functor interfaces
static __device__ __forceinline__ bool
    AdvanceFunctor(
  VertexId s_id,  // source node ID
  VertexId d_id,  // destination node ID
  DataSlice *d_data_slice,
  SizeT edge_id,  // edge list ID
  LabelT label, // label value
  SizeT input_pos,  // input queue idx
  SizeT &output_pos); // output queue idx
static __device__ __forceinline__ bool
    FilterFunctor(
  VertexId node,  // node ID
  DataSlice *d_data_slice,
  LabelT label, // label value
  SizeT input_pos,  // input queue idx
  SizeT output_pos);  // output queue idx

// operator interfaces

// advance
template<typename AdvancePolicy, typename
     Problem, typename Functor, typename
    AdvanceType>
__global__ void Advance(
SizeT queue_length,
VertexId *input_frontier,
VertexId *output_frontier,
DataSlice *data_slice);
```

```
// filter
template<typename FilterPolicy, typename
    Problem, typename Functor, bool
    ComputeFlag>
__global__ void Filter(
SizeT queue_length,
VertexId *input_frontier,
VertexId *output_frontier,
DataSlice *data_slice);

// neighborhood reduction
template<typename NeighborhoodPolicy,
    typename Problem, typename Functor,
    typename ReduceOp>
__global__ void Neighborhood(
SizeT queue_length,
VertexId *input_frontier,
VertexId *output_frontier,
DataSlice *data_slice);

// segmented intersection
template<typename IntersectionPolicy,
    typename Problem, typename Functor>
__global__ long Intersection(
SizeT queue_length,
VertexId *input_frontier1,
VertexId *input_frontier2,
VertexId *output_frontier,
DataSlice *data_slice);
```

Figure 2.5: *Gunrock's Graph Operator and Functor APIs.*

Gunrock's software architecture is divided into two parts. Above the traversal -compute abstraction is the application module. This is where users define different graph primitives using the high-level APIs provided by Gunrock. Under the abstraction are the utility functions, the implementation of operators used in traversal, and various optimization strategies.

Gunrock programs specify three components: the *problem*, which provides graph topology data and an algorithm-specific data management interface; the *functors*, which contain user-defined computation code and expose kernel fusion opportunities that we discuss below; and an *enactor*, which serves as the entry point of the graph algorithm and specifies the computation as a series

of graph operator kernel calls with user-defined kernel launching settings.

Given Gunrock's abstraction, the most natural way to specify Gunrock programs would be as a sequence of bulk-synchronous steps, specified within the enactor and implemented as kernels, that operate on frontiers. Such an enactor is in fact the core of a Gunrock program, but an enactor-only program would sacrifice a significant performance opportunity. We analyzed the techniques that hardwired (primitive-specific) GPU graph primitives used to achieve high performance. One of their principal advantages is leveraging producer-consumer locality between operations by integrating multiple operations into single GPU kernels. Because adjacent kernels in CUDA or OpenCL share no state, combining multiple logical operations into a single kernel saves significant memory bandwidth that would otherwise be required to write and then read intermediate values to and from memory. The CUDA C++ programming environment we use has no ability to automatically fuse neighboring kernels together to achieve this efficiency (Chapter 3 will provide more detailed discussion on kernel fusion and other types of optimizations that fit in Gunrock's data-centric programming model).

We summarize the interfaces for these operations in Figure 2.5. Our focus on kernel fusion enabled by our API design is absent from other programmable GPU graph libraries, but it is crucial for performance. Chapter 4 will provide more detailed discussion on Gunrock's framework design and implementation.

To conclude this chapter, we list the benefits of Gunrock's data-centric programming model for graph processing on the GPU:

- The data-centric programming model allows more flexibility on the operations, since frontier is a higher level abstraction of streaming data, which could represent nodes, edges, or even arbitrary sub-graph structures. In Gunrock, every operator can be vertex-centric or edge-centric for any iteration. As a result, we can concentrate our effort on solving one problem: implementing efficient operators, and see that effort reflected in better performance on various graph primitives.

- The data-centric programming model decouples compute operator from traversal opera-

tors. For more generalized data-parallel primitives such as prefix-sum and reduce, it is reasonable to integrate the computation with operation due to the limited choice and complexity of operations. However, for different graph primitives, computations can be very different in terms of complexity and the way it interacts with graph operators. Decoupling compute operator from traversal operators gives Gunrock more flexibility in presenting various graph primitives with the framework of doing customized computation on a set of operations that abstract most operations appear in graph primitives.

- The data-centric programming model builds on top of state-of-the-art data -parallel primitives and enables various types of optimizations. This makes data-centric programming model have better performance than any other GPU graph processing systems and achieve comparable performance to specialized GPU graph algorithms.

- The data-centric programming model uses high level graph operator interfaces to encapsulate complicated implementations. This makes Gunrock's application code smaller in size and clearer in logic compared to other GPU graph libraries. Gunrock's Problem class and kernel enactor are both template-based C++ code; Gunrock's functor code that specifies per-node or per-edge computation is C-like device code without any CUDA-specific keywords.

- The data-centric programming model easies the job of expanding single-GPU execution model. It fits with other execution models such as semi-asynchronous execution model and multi-GPU single-node/multi-node execution model. Our multi -GPU graph processing framework [65] is built on top of our data-centric programming model with unchanged core single-GPU implementation and advanced communication and partition module designed specifically for multi-GPU execution.

Chapter 3

GPU Graph Analytics System Implementation and Optimizations

The implementation of efficient graph operators is critical to Gunrock's performance. This chapter discusses how to build these efficient graph operators using basic data-parallel primitives, and further describes how to encapsulate multiple optimization strategies into our design.

3.1 Efficient Graph Operator Design

Section 1.3.3 shows all the data-parallel primitives we need to build the five graph operators in Gunrock. In this section, we analyze the implementation of each graph operator and briefly discuss how these implementations enable optimizations which we will describe in more details in Section 3.2. Figure 3.1 shows the sample graph we use to show our graph operators design and figure 3.2 shows the corresponding arrays for graph storage in compressed sparse row (CSR) format (discussed in section 3.2.4) in Gunrock.

3.1.1 Advance

Advance operator takes the input frontier, visits the neighbor list of each item in the input frontier, then writes the output neighbor lists in the output frontier. An efficient implementation

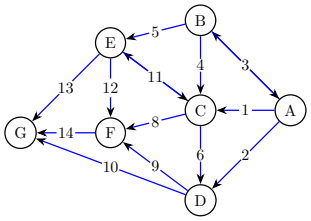


Figure 3.1: *A sample directed graph with 7 nodes and 15 edges.*

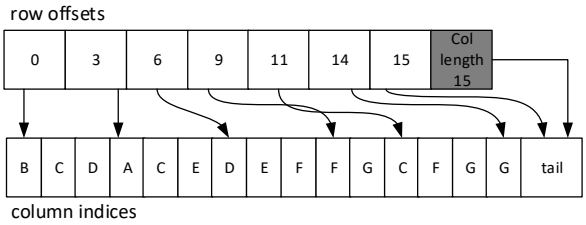


Figure 3.2: *CSR format of sample graph in Gunrock.*

of advance needs to reduce the task granularity to a homogenous size and then evenly distribute these smaller tasks among threads [59] so that we can parallelize the process.

At a high level, advance can be seen as a vectorized device memory allocation and copy, where parallel threads place dynamic data (neighbor lists with various lengths) within shared data structures (output frontier). The efficient parallelization of this process requires two things: for the allocation part, given a list of allocation requirements for each input item (neighbor list size array computed from row offsets), we need the scatter offsets to write the output frontier; for the copy part, we need to load balance parallel scatter writes with various lengths over a single launch.

The first part is typically implemented with prefix-sum. For the second part, ther are several different implementations. These implementations differ in the following ways:

load balancing: A coarse-grained load balancing strategy will divide the neighbor lists into three groups according to two size threshold values, a fine-grained one will either map the same number of input items or the same number of output items to a group of threads.

traversal direction: A push-based advance will expand the neighbor lists of current input frontier, a pull-based advance will intersect the neighbor lists of unvisited node frontier with the current frontier.

We will provide more details in section 3.2.1.

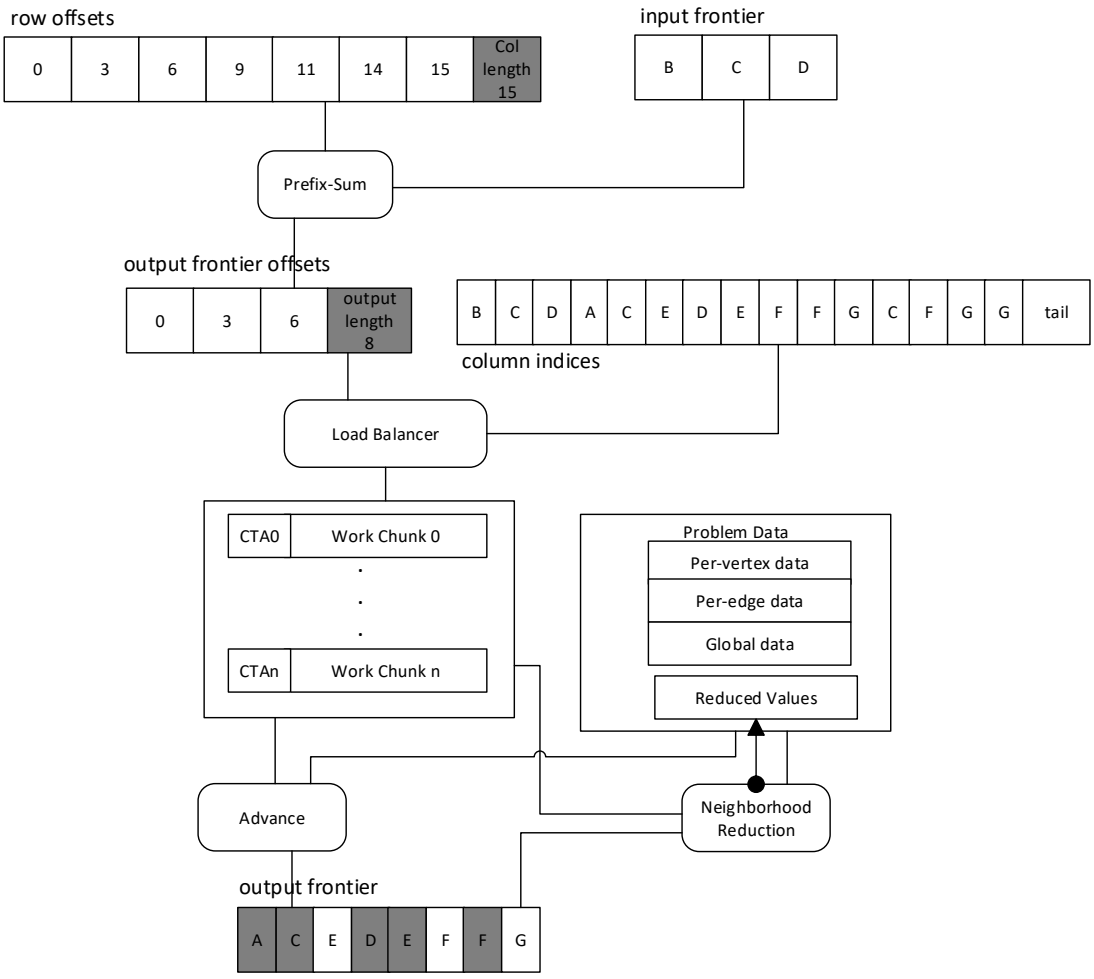


Figure 3.3: *Both advance and neighborhood reduction have a prefix-sum part and a load balancer part, neighborhood reduction will additionally compute a segmented reduction per neighbor list.*

3.1.2 Neighborhood Reduction

Neighborhood reduction operator takes the input frontier, visits the neighbor list of each item in the input frontier and performs a user-specified reduction over each neighbor list before writes the output items in the output frontier. An efficient implementation of neighborhood reduction involves dynamically generating a segmented neighborhood array and performing parallel segmented reduction over each segment.

The key to the efficiency of the neighborhood reduction operator also relies on the parallel allocation of the workload and the load balanced workload mapping. The internal pipeline of

neighborhood reduction is also parallel pre-allocation using prefix-sum, and load balance parallel segmented reduction. The second part could reuse both workload mapping strategies in advance with minor modification for doing segmented reduction. Figure 3.3 shows the implementation details of these two operators.

3.1.3 Filter

Filter operator is in essence a stream compaction operator that transforms a sparse presentation of array (input frontier) to a compact one (output frontier) where the sparsity comes from the different returned values for each input item in the validity test function (for graph traversal, multiple redundant nodes will fail the validity test). An efficient stream compaction implementation also relies on prefix-sum, it is a well-solved problem [8, 36]. In Gunrock, we adopt Merrill et al.'s filtering implementation based on local prefix-sum and various culling heuristics. The byproduct of this implementation is a uniquification feature that does not strictly require the complete removal of undesired items in the input frontier. More details will be discussed in Section 3.2.2.

3.1.4 Segmented Intersection

Segmented intersection operator takes two input frontiers and for each pair of input items, computes the intersection of two neighbor lists, and outputs the segmented items. It is known that for intersection computation on two large frontiers, a modified merge-path algorithm would achieve high performance because of its load balance [5]. However, for segmented intersection, the workload per input item pair depends on the size of each item's neighbor list. For this reason, we still use prefix-sum for pre-allocation, then perform a series of load-balanced intersections according to a heuristic based on the sizes of the neighbor list pairs. Finally, we use a stream compaction to generate the output frontier, and a segmented reduction as well as a global reduction to compute segmented intersection counts and the global intersection count.

High-performance segmented intersection requires a similar focus as high-performance graph traversal: effective load-balancing and GPU utilization. In our implementation, we use the

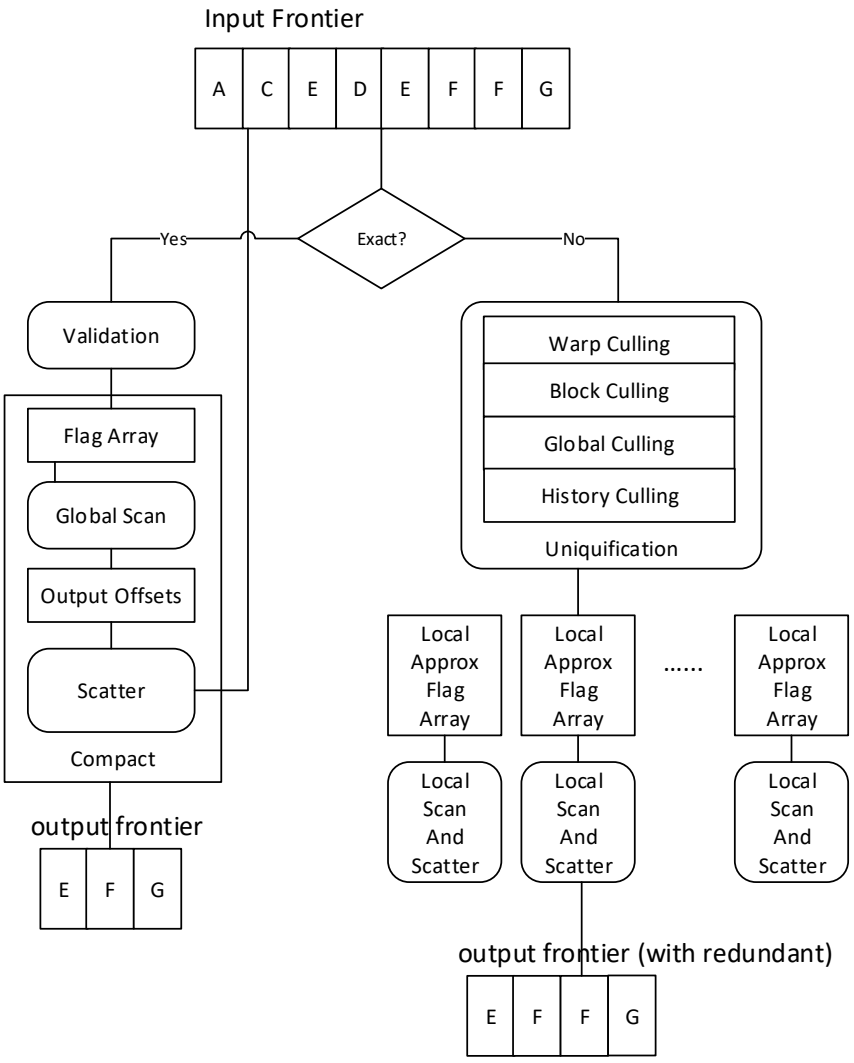


Figure 3.4: *Filter is based on compact, which uses either a global scan and scatter (for exact filtering) or a local scan and scatter after heuristics (for inexact filtering).*

same dynamic grouping strategy proposed in Merrill's BFS work [59]. We divide the edge lists into two groups: (1) small neighbor lists; and (2) one small and one large neighbor list. We implement two kernels (Two-Small and Small-Large) that cooperatively compute intersections. Our TwoSmall kernel uses one thread to compute the intersection of a node pair. Our SmallLarge kernel starts a binary search for each node in the small neighbor list on the large neighbor list. By using this 2-kernel strategy and carefully choosing a threshold value to divide the edge list into two groups, we can process intersections with same level of workload together to gain load balancing and higher GPU resource utilization. Currently, if two neighbor lists of a pair are both large neighbor list, we will still use SmallLarge kernel where ideally a third kernel that utilize

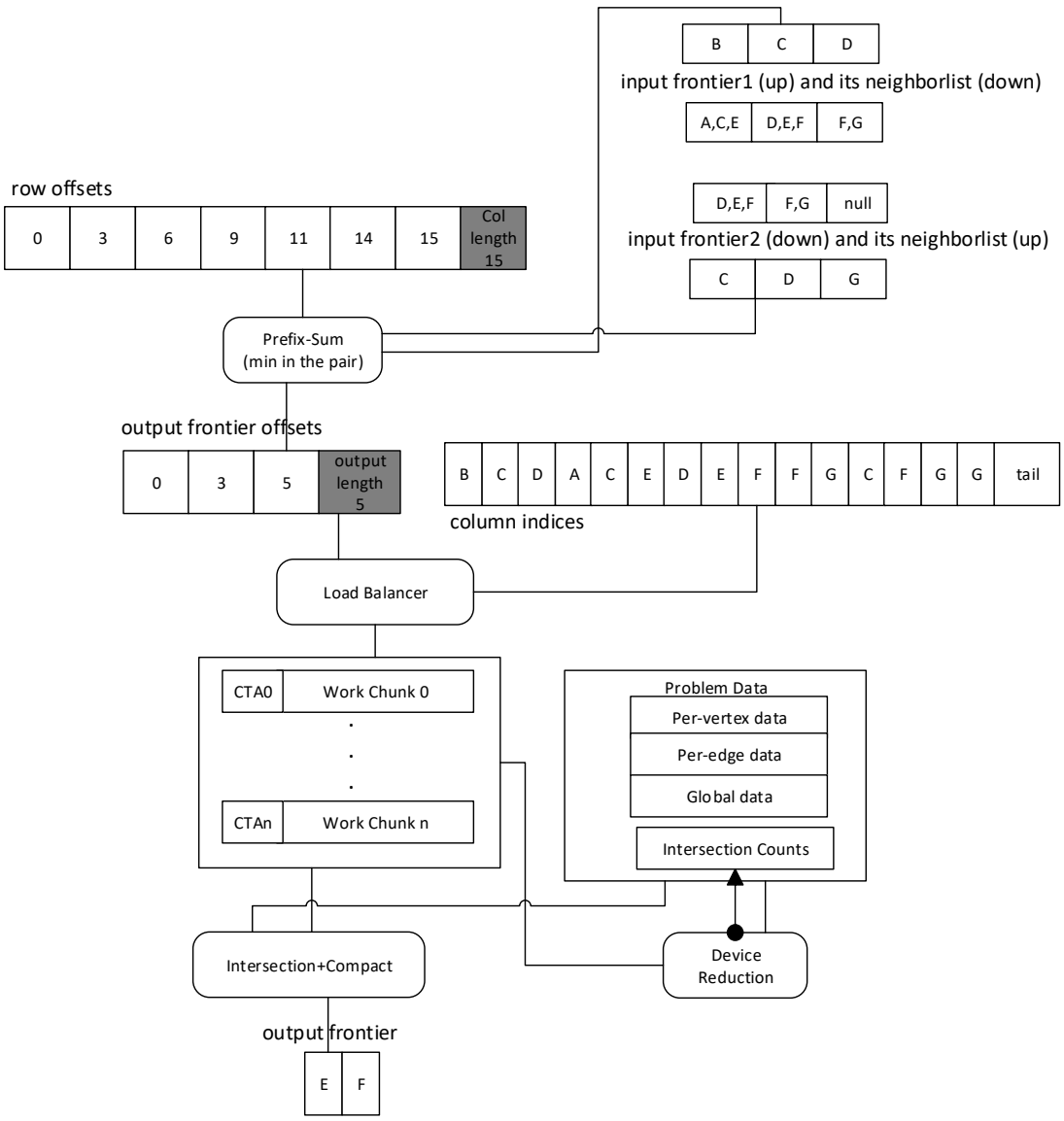


Figure 3.5: *Segmented intersection uses all the data parallel primitives we discussed in Section 1.3.3 including prefix-sum, compact, merge-based intersection, and reduction.*

all threads within a block to work on two large neighbor lists will yield better performance.

3.2 Optimizations for Graph Analytics on the GPU

Choosing the right abstraction is one key component in achieving high performance within a graph framework. The second component is optimized implementations of the primitives within the framework. One of the main goals in designing the Gunrock abstraction was to easily allow integrating existing and new alternatives and optimizations into our primitives to give more options to programmers. In general, we have found that our data-centric abstraction, and our focus on manipulating the frontier, have been an excellent fit for these alternatives and optimizations, compared to a more difficult implementation path for other GPU computation-focused abstractions. This section, we offer examples by discussing optimizations that help increase the performance in four different categories:

- Graph traversal throughput;
- Synchronization throughput;
- Kernel launch throughput;
- Memory access throughput.

3.2.1 Graph Traversal Throughput Optimizations

One of Gunrock's major contributions is generalizing different types of workload-distribution and load-balance strategies that previously only applied to specialized GPU graph primitives into Gunrock's general-purpose advance operator so that it can improve the performance for any graph primitive that uses advance operator.

In this section, we define workload for graph primitives as per-edge and/or per-node computation that happens during graph traversal. Gunrock's advance step generates an irregular workload. Consider an advance that generates a new vertex frontier from the neighbors of all vertices in

Optimization Strategy	Module Name in Gunrock
Static Workload Mapping	ThreadExpand
Dynamic Grouping Workload Mapping	TWC_FORWARD
Merge-based Load-Balanced Partitioning Workload Mapping	LB, LB_LIGHT, and LB_CULL
Pull Traversal	LB_BACKWARD and TWC_BACKWARD

Table 3.1: *Four graph traversal throughput optimization strategies and their corresponding module names in Gunrock implementation, where LB_LIGHT processes load balance over input frontier, LB processes load balance over output frontier, and LB_CULL combined LB and LB_LIGHT with a follow-up filter into a fused kernel.*

the current frontier. If we parallelize over input vertices, graphs with a variation in vertex degree (with different-sized neighbor lists) will generate a corresponding imbalance in per-vertex work. Thus, mapping the workload of each vertex onto the GPU so that they can be processed in a load-balanced way is essential for efficiency. Table 3.1 shows our traversal throughput optimization strategies and their corresponding module name in Gunrock implementation. Later in this section, we summarize the pros and cons and the guideline for usage of these optimizations in table 3.2.

Algorithm 2 Static Graph Traversal Workload Mapping

Input:
G ▷ Csr format of graph storage
I ▷ The input frontier
Output:
O ▷ The output frontier

```
1: row_offsets ← G.rowoffsets
2: output_indices ← PrefixSum(I, row_offsets)
3: parfor sid ← 0, len(I) do
4:     output_idx ← output_indices[sid]
5:     offset_begin ← row_offsets[I(sid)]
6:     offset_end ← row_offsets[I(sid) + 1]
7:     for col_id ← offset_begin, offset_end do
8:         did ← G.colindices[col_id]
9:         Process(sid, did, P)
10:        O[output_indices] ← did
11:    end for
12: end parfor
```

The most significant previous work in this area balances load by cooperating between threads. Targeting BFS, Hong et al. [37] map the workload of a single vertex to a series of virtual warps. Merrill et al. [59] use a more flexible strategy which maps the workload of a single vertex to a thread, a warp, or a block, according to the size of its neighbor list. Targeting SSSP, Davidson et al. [18] use two load-balanced workload mapping strategies, one that groups input work and the other that groups output work. The first does load balance over the input frontier, the second does load balance over the output frontier. Every strategy has its tradeoff on computation overhead and load balancing performance. We show in Gunrock how we integrate and redesign these two strategies, use heuristics to choose from them to achieve the best performance for different datasets, and generalize this optimization to various graph operators in Gunrock, such as advance, neighborhood reduction, and segmented intersection.

3.2.1.1 Static Workload Mapping Strategy

One straightforward approach to map the workload is to map one frontier vertex's neighbor list to one thread. Each thread loads the neighbor list offset for its assigned node, then serially

processes edges in its neighbor list. Algorithm 2 shows such per-thread workload mapping strategy without optimization. Though this simple solution will cause severe load imbalance for scale-free graphs with unevenly distributed degrees and extremely small diameter, it has its advantages of negligible load balancing overhead and works well for large-diameter graphs with a relatively even degree distribution. Thus in Gunrock, we keep this static strategy but provide several improvements in the following aspects:

cooperative process: we load all the neighbor list offsets into shared memory, then use a block of threads to cooperatively process per-edge operations on the neighbor list.

loop strip mining: we split the neighbor list of a node so that multiple threads within the same SIMD lane could achieve better utilization.

We found out that this method performs better when used for large-diameter graphs with a relatively even degree distribution since it balances thread work within a block, but not across blocks. For graphs with a more uneven degree distribution (e.g., scale-free social graphs), we turn to a second strategy.

3.2.1.2 Dynamic Grouping Workload Mapping Strategy

Gunrock's TWC strategy is based on Merrill et al's BFS implementation [59], with more flexible launch settings and user-specific computation functor support so that as a graph operator building block, it can be generalized to traversal steps in other traversal-based graph primitives. It is implemented using persistent threads technique to solve the performance bottleneck of our ThreadExpand method due to significant differences of neighbor list size. Like Merrill et al., we directly address the variation in size by grouping neighbor lists into three categories based on their size, then individually processing each category with a strategy targeted directly at that size in a persistent threads programming style. Our three sizes are (1) lists larger than a block; (2) lists larger than a warp (32 threads) but smaller than a block; and (3) lists smaller than a warp. We begin by assigning a subset of the frontier to a block. Within that block, each thread owns one node. The threads that own nodes with large lists arbitrate for control of the entire block. All the threads in the block then cooperatively process the neighbor list of the winner's node.

This procedure continues until all nodes with large lists have been processed. Next, all threads in each warp begin a similar procedure to process all the nodes whose neighbor lists are medium-sized lists. Finally, the remaining nodes are processed using our ThreadExpand method. As Merrill et al. noted in their paper, this strategy can guarantee a high utilization of resource and limit various types of load imbalance such as SIMD lane underutilization (by using per-thread mapping), intra-thread imbalance (by using warp-wise mapping), and intra-warp imbalance (by using block-wise mapping). (Figure 3.6).

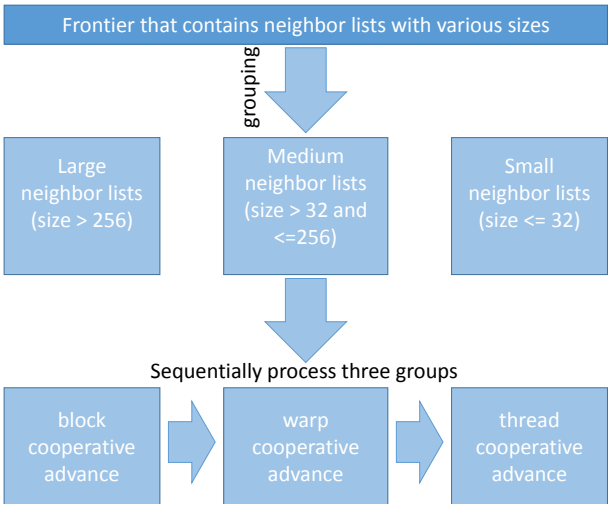


Figure 3.6: *Dynamic grouping workload mapping strategy [59].*

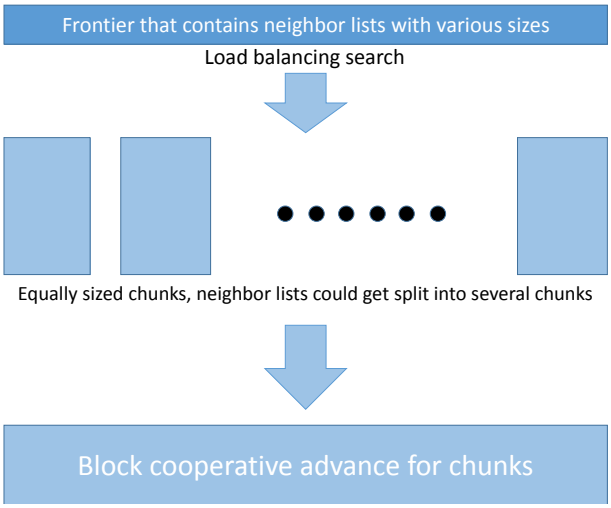


Figure 3.7: *Merge-based load-balanced partitioning workload mapping strategy [18].*

3.2.1.3 Merge-based Load-Balanced Partitioning workload mapping strategy

The specialization of dynamic grouping workload mapping strategy allows higher throughput on frontiers with a high variance in degree distribution, but at the cost of higher overhead due to the sequential processing of the three different sizes. Also, to deal with intra-block load imbalance, additional scheduling and work stealing method needs to be applied and will further add to the load balancing overhead. A global one-pass load balancing strategy would potentially solve the intra-block load imbalance problem and thus bring better performance. Davidson et al. and Gunrock improve on the dynamic grouping workload mapping strategy by globally load balancing over either input frontier or output frontier. This introduces load balancing overhead, but significantly increases the traversal throughput for scale-free graphs.

input frontier load balance maps the same number of input items to a block, then put the output offset for each input item computed by prefix-sum into shared memory. Cooperative process and loop strip mining are both used here too. All threads within a single block will cooperatively visit all the neighbor lists of the input items that belong to this block, when a thread starts process a new neighbor list, it needs a binary search to find the corresponding source node ID.

output frontier load balance first uses a global prefix-sum and to compute all the output offsets, then forms an arithmetic progression of $0, N, 2N, ..., |Input|$ where N is the number of edges each block processes. A global sorted search [6] of this arithmetic progression in the output offset array will find the starting indices for all the blocks within the frontier. After organizing groups of edges into equal-length chunks. All threads within one block cooperatively process edges, when we start to process a neighbor list of a new node, we use binary search to find the node ID for the edges that are going to be processed. Using this method, we ensure both inter-and-intra-block load-balance. (Figure 3.7).

At the high level, Gunrock makes a load-balancing strategy decision depending on topology. We note that our load-balancing workload mapping method performs better on social graphs with irregular distributed degrees, while the dynamic grouping method is superior on graphs where most nodes have small degrees. For this reason, in Gunrock we implement a hybrid of both methods on both vertex and edge frontiers, using the dynamic grouping strategy for nodes with relatively smaller neighbor lists and the load-balancing strategy for nodes with relatively larger neighbor lists. We pick average degree as the metrics for choosing between these two strategies, when the graph has an average degree of 5 or larger, we use load-balancing strategy, otherwise we use dynamic grouping strategy. Within the load-balancing strategy, we set a static threshold. When the frontier size is smaller than the threshold, we use coarse-grained load-balance over nodes (input frontier), otherwise coarse-grained load-balance over edges (output frontier). We have found that setting this threshold to 4096 yields consistent high performance for tests across all Gunrock-provided graph primitives. Users can also change this value easily in the Enactor module for their own datasets or graph primitives. Superior load balancing is one of the most

significant reasons why Gunrock outperforms other GPU frameworks [90].

3.2.1.4 Push vs. Pull Traversal

Other GPU programmable graph frameworks also support an advance step, of course, but because they are centered on vertex operations on an implicit frontier, they generally support only "push"-style advance: the current frontier of active vertices "pushes" active status to its neighbors to create the new frontier. Beamer et al. [7] described a "pull"-style advance on CPUs: instead of starting with a frontier of active vertices, pull starts with a frontier of *unvisited* vertices, generating the new frontier by filtering the unvisited frontier for vertices that have neighbors in the current frontier.

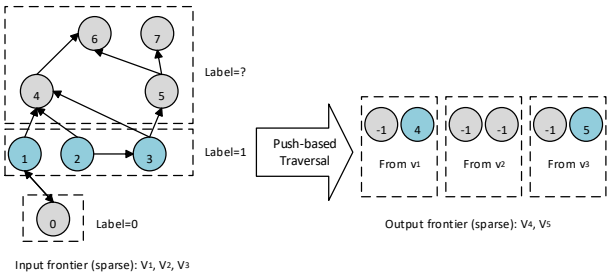


Figure 3.8: *Push-based graph traversal.*

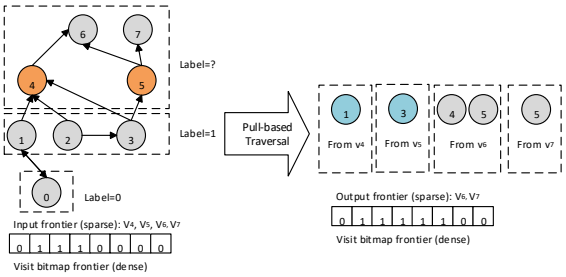


Figure 3.9: *Pull-based graph traversal.*

Beamer et al. showed this approach is beneficial when the number of unvisited vertices drops below the size of the current frontier. While vertex-centered GPU frameworks have found it challenging to integrate this optimization into their abstraction, our data-centric abstraction is a much more natural fit because we can easily perform more flexible operations on frontiers. Gunrock internally converts the current sparse frontier presented in compact form into a dense bitmap frontier to show the visited status of each vertex in the graph, then generates a new frontier which contains all the unvisited nodes, and uses an advance step to "pull" the computation from these nodes' predecessors if they are valid in the bitmap. The capability of keeping two active frontiers (in this case one bitmap frontier for visited status check, and one normal frontier of unvisited nodes) differentiate Gunrock from all other GPU graph processing programming models.

Algorithm 3 Generalized advance algorithm

Input:
G ▷ Csr format of graph storage
I ▷ The input frontier
Output:
O ▷ The output frontier

```
1: row_offsets ← G.rowoffsets
2: output_indices ← PrefixSum(I, row_offsets)
3: if output_indices[len(I)] < threshold then
4:     if advance_mode = pull then
5:         F ← GenFrontier(G.labels, iteration_num, V)
6:     else
7:         F ← I
8:     end if
9:     Advance(G, F, O)
10: else
11:     bitmap ← SparseToDense(G.nodes, I)
12:     U ← GenUnvisited(G.labels, V)
13:     Reverse_Advance(G, bitmap, U, O)
14: end if
```

For pull-based traversal on the GPU, there are two observations we have made:

Compacted unvisited frontier shows better performance: Unlike CPU's quadratic pull-based traversal implementations [7, 78], which always check all the vertices and trigger advance when the vertex is unvisited, Gunrock's data-centric programming model could keep an unvisited frontier in a ping-pong buffer in compact form to implement a linear traversal for better memory access throughput.

Heuristic of switching between push and pull-based traversal: Beamer et al. use two heuristics to switch between push and pull-based traversal on shared memory CPU system. Given the number of edges to check from the frontier (m_f), the number of vertices in the frontier (n_f), the edges to check from unexplored vertices (m_u), and two tuning parameters α and

β, they define two equations:

$$m_f > \frac{m_u}{\alpha} = C_{TB} \tag{3.1}$$

$$n_f < \frac{n}{\beta} = C_{BT} \tag{3.2}$$

where n is the graph node number, C_{TB} and C_{BT} are threshold for push-to-pull-based traversal switch and pull-to-push-based traversal switch separately. However, on the GPUs, we update these two heuristics for two reasons: 1) computing m_f and m_u requires two additional passes of prefix-sum; 2) using a compacted unvisited frontier, the performance difference between push-and-pull-based traversal is negligible when the number of unvisited nodes gets very small. The updated heuristic switches the push-to-pull-based traversal point to:

$$n_f > \frac{n_u}{\alpha} = C_{TB} \tag{3.3}$$

where n_f is the current frontier length, and n_u is the number of unvisited nodes. By modifying α, this heuristic could still find the optimal iteration to switch. As for the second switch, we found that on the GPU, for both scale-free graphs and road networks, setting β to a smaller number such that the second switch never happens will always bring better performance than the second switch.

With this optimization, we see a speedup on BFS of 1.52x for scale-free graphs and 1.28x for small-degree-large-diameter graphs. In an abstraction like Medusa, with its fixed method (segmented reduction) to construct frontiers, it would be a significant challenge to integrate a pull-based advance.

In data-centric programming model, we implement the pull-based traversal using the same interface as push-based traversal so that it can be generalized to other traversal-based graph primitives beyond BFS.

3.2.1.5 Priority Queue

A straightforward BSP implementation of an operation on a frontier treats each element in the frontier equally, i.e., with the same priority. Many graph primitives benefit from prioritizing

Optimization Strategy	Summary
Static Workload Mapping	Low cost for load balancing; bad for varying degree distributions. Should use to traverse regular graphs.
Dynamic Grouping Workload Mapping	Moderate cost for load balancing; bad for scale-free graphs. Should use to traverse mesh-like graphs with large diameter when frontier size gets larger than 1 million.
Merge-based Load-Balanced Partitioning Workload Mapping	High cost for load balancing on power-law graphs, but low cost for regular graphs; shows consistant better performance on most graphs. Should use as a default traversal strategy choice.
Pull Traversal	Has one-time frontier conversion/preparation cost; good for scale-free graphs with large amount of common neighbors. Should not use on regular graphs and when the number of unvisited vertices is either too large or too small (For more details please refer to section 6.2).

Table 3.2: *Four graph traversal throughput optimization strategies, their pros and cons and guideline for usage in Gunrock.*

certain elements for computation with the expectation that computing those elements first will save work overall (e.g., delta-stepping for SSSP [60]). Gunrock generalizes the approach of Davidson et al. [18] by allowing user-defined priority functions to organize an output frontier into "near" and "far" slices. This allows the GPU to use a simple and high-performance split operation to create and maintain the two slices. Gunrock then considers only the near slice in the next processing steps, adding any new elements that do not pass the near criterion into the far slice, until the near slice is exhausted. We then update the priority function and operate on the far slice.

Like other graph operators in Gunrock, constructing a priority queue directly manipulates the frontier data structure. It is difficult to implement such an operation in a GAS-based programming model since that programming model has no explicit way to reorganize a frontier.

The implementation of priority queue in Gunrock is a modified filter operator, where it uses two

stream compactions to not only form the output frontier of input items with true flag values, but also form a "far" pile of input items with false flag values.

Currently Gunrock uses this specific optimization only in SSSP, but we believe a workload reorganization strategy based on a more general multisplit operator [2] that has maps one input frontier to multiple output frontiers according to arbitrary number of priority levels would fit nicely into Gunrock's many-frontiers-in, many-frontiers-out data-centric programming model. Such a generalized strategy could enable a semi-asynchronous execution model in Gunrock since different frontiers in a multi-frontier set can process an arbitrary number of BSP steps. This will potentially increase the performance of various types of node ranking, community detection, and label propagation algorithms as well as algorithms on graphs with small "long tail" frontiers.

3.2.2 Synchronization Throughput Optimization

For graph processing on the GPU, the bottlenecks of synchronization throughput come from two places:

concurrent discovery: In a tree traversal, there can be only one path from any node to any other node. However in graph traversal, starting from a source node, there could be several redundant visits if no pruning is conducted. Beamer et al. [7] categorize these redundant visits into three types: 1) visited parents; 2) being visited peers; and 3) concurrent discovered children. Concurrent discovery of child nodes contribute to most synchronization overhead when there is per-node computation.

dependencies in parallel data-primitives: Sometimes the computation during traversal has a reduction (Pagerank, BC) and/or intersection (TC) step on each neighbor list.

For synchronization overhead brought by concurrent discovery, the generalized pull-and-push based traversal discussed in section 3.2.1.4 will reduce it. This section we present another optimization that solves this problem from the perspective of the idempotence of a graph operation.

3.2.2.1 Idempotent vs. non-idempotent operations

Multiple elements in the frontier may share a common neighbor, this has two consequences: 1) it will cause an advance step to generate an output frontier that has duplicated elements. 2) it will cause the computation on the common neighbors (if any) to perform multiple times. The second consequence will cause potential synchronization problem by producing race conditions. A general way to avoid race conditions is to introduce atomic operations. However, for some graph primitives with "idempotent" operations (e.g., BFS's visit status check), repeating a computation causes no harm, Gunrock's advance step will avoid atomic operations, repeat the computation multiple times, and output all the redundant items to the output frontier. Gunrock's filter step has incorporated a series of inexpensive heuristics [59] to reduce, but not eliminate, redundant entries in the output frontier. These heuristics include a global bitmask hashtable, a block level history hashtable, and a warp level hashtable. The sizes of each hashtable is adjustable to achieve the optimal tradeoff between performance and redundancy reduce rate.

3.2.2.2 Atomic Avoidance Reduction Operations

To reduce the synchronization overhead brought by reduction, we either reduce the atomic operations by hierarchical reduction and the efficient use of shared memory on the GPU, or assign several neighboring edges to one thread in our dynamic grouping strategy so that partial results within one thread can be accumulated without atomic operations.

3.2.3 Kernel Launch Throughput Optimization

Gunrock and all other BSP model-based GPU graph processing library launch one or several GPU kernels per iteration, and often copy a condition check byte from device to host after each iteration to decide whether to terminate the program. Pai and Pingali noted [64] this kernel launch overhead and proposed several compiler optimizations to reduce it. In Gunrock we also proposed two optimizations in our framework:

Fuse computation with graph operator: Specialized GPU implementations fuse regular computation steps together with more irregular steps like advance and filter by running a com-

putation step (with regular parallelism) on the input or output of the irregularly-parallel step, all within the same kernel. To enable similar behavior in a programmable way, Gunrock exposes its computation steps as *functors* that are integrated into all its graph operator kernels at compile time to achieve similar efficiency. We support functors that apply to {edges, vertices} and either return a Boolean value (the "cond" functor), useful for filtering, or perform a computation (the "apply" functor). These functors will then be integrated into Gunrock's graph operator kernel calls, which hide any complexities of how those steps are internally implemented.

Fuse filter step with traversal operators: Several traversal-based graph primitives will have a filter step immediately after the advance or neighborhood reduction step, Gunrock integrates the fused traversal operator which launches advance and filter steps within one kernel. The additional benefit of doing this is to reduce the data movements between double-buffered input and output frontier.

3.2.4 Memory Access Throughput Optimization

For graph problems that require irregular data accesses, in addition to exposing enough parallelism, a successful GPU implementation benefits from the following application characteristics: 1) coalesced memory access, 2) effective use of the memory hierarchy, and 3) reducing scattered reads and writes.

In terms of graph storage format, Gunrock uses a compressed sparse row (CSR) sparse matrix for vertex-centric operations by default and allow users to choose an coordinate list (COO) representation for edge-centric operations. CSR uses a column-indices array, C, to store a list of neighbor vertices and a row-offsets array, R, to store the offset of the neighbor list for each vertex. It provides compact and efficient memory access, and allows us to use prefix-sum to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph processing [59]. In terms of data structures, Gunrock represents all per-node and per-edge data as structure-of-array (SOA) data structures that allow coalesced memory accesses with minimal memory divergence. In terms of graph operators, Gunrock always uses carefully designed

addressing for loops in kernel to guarantee coalesced memory access. We also efficiently use shared memory and local memory to increase memory access throughput.

In dynamic grouping workload mapping: Gunrock moves chunks of input frontier into local memory, and uses warp scan and warp streaming compaction.

In load-balanced partition workload mapping: Gunrock uses shared memory to store the resulting indices computed by merge-based load balanced search.

In filter operator: Gunrock stores two types of hash tables (block-wise history hashtable and warp-wise history hashtable) in the shared memory.

Chapter 4

Mini Gunrock: A Lightweight Graph Analytics Framework on the GPU

The purpose of Gunrock is to develop a programmable graph processing system with high performance. Our implementation is based on a data-centric programming model that increases flexibility and programmability over other GPU graph processing libraries and achieves comparable performance with specialized GPU graph algorithms. However, there is still an important question we need to answer: What is the right level of abstraction of a graph analytics programming model on the GPU? Our data-centric programming model focuses on evolving frontiers when running a graph primitive. One iteration could be generalized as an equation:

$$f_{out} = Op(f_{in}, G, P) \tag{4.1}$$

where f_{in} and f_{out} are the input and output frontiers respectively, $Op(f, G, P)$ is a graph operation on frontier f using data from graph G and per-node or/and per-edge data P. Gunrock implements each graph operator with specific low-level optimizations. While implementing the system, we noticed that there are several common data-parallel primitives used in different graph operators. We categorize these operators into three groups:

Per-Item Computation: simple regular kernels;

Prefix-Sum Based: prefix-sum and streaming compaction;

Irregular Computation: load-balanced search and segmented reduction.

With this finding in mind, we take another approach to implementing Gunrock's data-centric programming model: building graph operators on top of lower-level common data-parallel primitives. We call resulting lightweight graph analytics framework "Mini Gunrock". We have three design goals for Mini Gunrock:

Flexibility: To have the same features for graph operators in Gunrock;

Simplicity: To enable fast development of new graph operators and primitives with less and cleaner code;

Comparable Performance: To have comparable performance with Gunrock.

To achieve the above goals, we need to use a set of lower-level data-parallel primitives that have a certain level of flexibility and offer state-of-the -art performance. Moderngpu 2.0 has implemented a set of data-parallel primitives based on transform; each of them can take user-specified functors but otherwise inherit the high performance offered by moderngpu 1.0. We implement Mini Gunrock on top of moderngpu's transform-based primitives.

4.1 Transform-based Data-Parallel Primitives

The transform algorithm is one of the mutating sequence operations in C++ standard. Without losing generality, in this dissertation we refer to transform as the unary transform algorithm that applies a unary operation to every element in a range of input iterators exactly once. Transform is a mutating algorithm; it could modify input items, and it returns an iterator to the end of the output range. Compared with the graph operators we defined in section 3.1, transform is more flexible as it can define both regular and irregular tasks. Moderngpu 2.0 has implemented most of the data-parallel primitives we defined above using parallel transforms. These transforms include: *scan_transform* (parallel transform for prefix-sum), *lbs_transform* (parallel transform for merge-based load balanced search), *segreduce_transform* (parallel transform for segmented reduction), and *compact_transform* (parallel transform for streaming compaction). *lbs_transform*

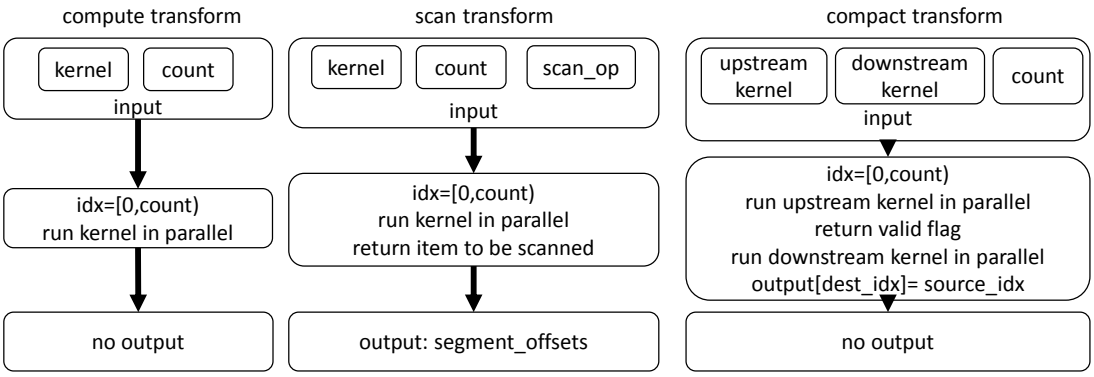


Figure 4.1: *Compute, scan and compact transforms in Mini Gunrock.*

is a parallel search of an array of sorted input items in an array of sorted output items, so that the input items can be then coordinated with their output items. It essentially turns an offset array and a neighbor list array into an edge list in parallel, which is a general implementation of Gunrock's merge-based load balanced partitioning workload mapping strategy (which we describe in section 3.2.1.3).

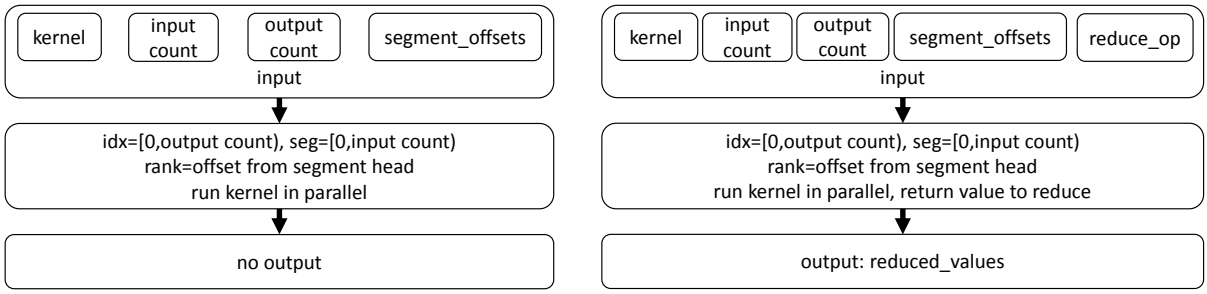


Figure 4.2: *LBS transform (left) and segreduce transform (right) in Mini Gunrock.*

Figure 4.1 and Figure 4.2 show the data flow of five transforms we use in Mini Gunrock. For every transform, users can define a named functor or a lambda function. Different transforms have different default auxiliary parameters for the functor. For the regular transform, scan transform, and compact transform, there is only one single auxiliary parameter *idx*, which implies the index of the input element. For lbs transform and segreduce transform, there are three auxiliary parameters *idx*, *seg*, and *rank*, where *idx* implies the index of the output frontier, *seg* implies the index of the input frontier, and *rank* implies the local offset value of the neighbor item. Using these auxiliary parameters and different combinations of these transforms, we could build most of our graph operators and some optimizations in Gunrock with a much smaller code size while

achieving comparable performance.

4.2 Graph Operator Implementation Using Transforms

Our goal of simplicity in Mini Gunrock has two aspects. The first one is a simple interface for using graph operators to develop quick prototypes for graph primitives. The second one is the simplicity of the graph operator implementation itself. Built on top of moderngpu transforms, our graph operator implementation in Mini Gunrock benefits from the high performance of moderngpu transforms, and yet has a clean interface.

Filter Operator Mini Gunrock's filter operator is implemented using *compact_transform*. The transform-based implementation has two phases: 1) an upsweep phase to mark flags (via a user-defined functor) for input items to be filtered, and 2) a downsweep using prefix-sum to generate the compacted output indices and write outputs in parallel.

Advance Operator The advance operator can be implemented using *scan_transform* and *lbs_transform*. The *scan_transform* on the neighbor list lengths of all the input items will generate an offsets array, which will serve as the beginning pointer of each input item's neighbor list in the output frontier. This array is then sent into *lbs_transform* to do the load-balanced search. Instead of having to implement sorted search, allocate shared memory, and handle communications between threads, like what Gunrock has implemented in its advance operator, *lbs_transform* automatically handles the workload mapping among all threads once given the offsets array, and generates the necessary information for the computation on each thread. As a major operator for all traversal-based graph primitives and several other graph primitives, making Mini Gunrock's advance efficient is one of the most critical aspects of its high performance. The lbs transform allows us to achieve that goal.

Neighborhood Reduction Operator Our neighborhood reduction operator can be implemented using *scan_transform* and *segreduce_transform*. In this operator, one pass of scan serves the same purpose as in advance. The following segmented reduction transform internally fuses the merge-based load-balanced search with the reduction computation. It has the

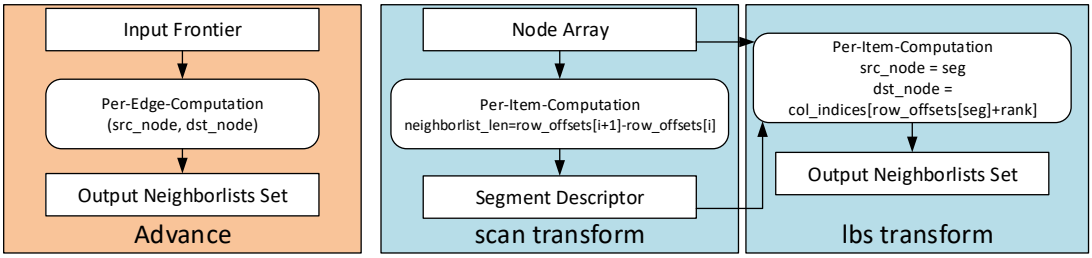


Figure 4.3: *The left block shows the input, output, and parallel operation of an advance operator in Mini Gunrock; the right block shows its implementation using transform-based primitives.*

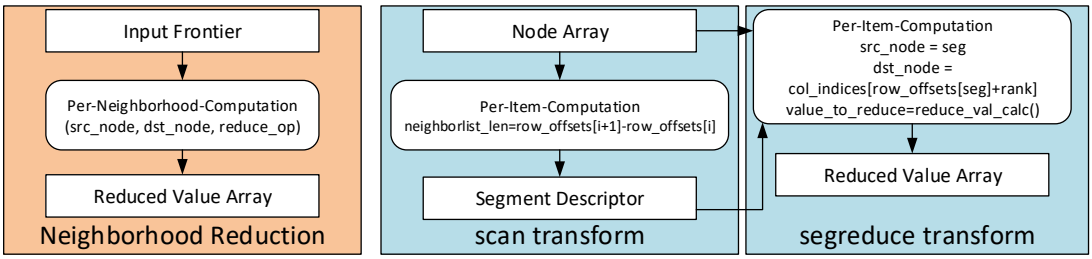


Figure 4.4: *The left block shows the input, output, and parallel operation of a neighborhood reduction operator in Mini Gunrockd. The, right block shows its implementation using transform-based primitives.*

same level of efficiency as the advance operator. We can easily choose whether to output the visited neighbor nodes, or to output a list of nodes computed by the reduction operator, depending on our next operation in the graph primitive. Also, we can set the neighborhood to be either out-degree neighbors (for a push-style neighborhood reduction) or in-degree neighbors (for a pull-style neighborhood reduction).

Compute Operator Our compute operator is merely a regular computation task on each input item. It is a basic *transform* with a user-defined functor as the computation step. The difference between *transform* and a specialized regular GPU kernel is that for *transform*, the algorithm will launch kernels with optimal settings that maximize utilization according to a set of GPU-architecture- and input-length-dependent pre-defined conditions.

4.2.1 Mini Gunrock vs. Gunrock

This section compares Mini Gunrock and Gunrock at the graph operator level.

As a more general type of primitive, transform-based primitives and combinations of them can

implement most of Gunrock's current graph operators. The exception is segmented intersection. However, this is due to the current moderngpu implementation, which only contains unary transforms; segmented intersection is a binary transform that takes two input arrays and produces one output array. It can use balanced path (a modified version of merge path) on the block level to produce an output array that contains all intersected items from two input arrays.

Compared to Mini Gunrock, Gunrock's graph operator implementation is more flexible, because Gunrock users have control of both launch settings and shared memory and local memory allocation. Gunrock's implementation makes advanced GPU techniques like persistent threads easier to implement. With such flexibility, users can also switch the optimization strategies used underneath the operator abstraction as needed. For neighborhood reduction, Gunrock's current implementation requires a separated segmented reduction pass after its advance, which is not optimal. However, it is possible to fuse this additional segmented reduction pass with the current advance kernel to increase performance.

4.3 Optimizations in Mini Gunrock

4.3.1 Advance-based Optimizations Using Transforms

Pull-based Traversal Mini Gunrock's pull-based implementation has two phases: data structure preparation and the actual pull-based advance. For data structure preparation, as in line 11 and 12 in algorithm 3, we use two transforms: one regular transform to generate a dense bitmap frontier representation from the sparse compacted frontier representation, then one *compact_transform* to generate a compacted frontier of unvisited nodes. For the pull-based advance, we use *scan_transform* and *lbs_transform* as in a push-based advance operator, but with a new functor to update the bitmap and process the computation on edge or destination vertices.

Flexible Features Compared to Gunrock, Mini Gunrock has various flexible features in advance and neighborhood reduction. These include: 1) for an input frontier that contains all of the edges in the graph, save the *scan_transform* pass to directly use the row offsets array;

2) for an advance operator whose output frontier is not used anywhere, skip writing to output frontier; and 3) for a neighborhood reduction, set direction to forward or backward to process the reduction on either out-degree neighbors or in-degree neighbors.

4.3.2 Filter-based Optimizations Using Transforms

Idempotence The idempotence optimization allows arbitrary heuristic filtering functors to be added in the upsweep pass of the *compact_transform*. An additional benefit of having this optimization in a filter operator is the avoidance of atomic operations in an advance operator when the computation is idempotent, with a tradeoff of potentially larger frontier size per iteration.

Two-level Priority Queue In Mini Gunrock, a two-level priority queue can be implemented with a *compact_transform* with a user-defined priority score function and threshold value.

4.3.3 Mini Gunrock vs. Gunrock

This section compares Mini Gunrock and Gunrock from the optimization perspective.

In Mini Gunrock, we have implemented several optimizations from Gunrock. The advance-based optimizations include pull-based traversal and flexible features such as optimization for all edge input frontiers, output frontier skipping, and neighborhood reduction on either in-degree neighbors or out-degree neighbors. The filter-based optimizations include idempotence and a two-level priority queue.

However, due to the different way Mini Gunrock and Gunrock process items within operators, the performance of these two optimizations differs between the two implementations. The redundancy-removal heuristics in Mini Gunrock are less effective compared to Gunrock, thus causing an explosion of frontier size, which slows down the convergence when used. This is because in Gunrock, after we decide the workload for each block, we use a smaller tile to move the value of these items from global memory to local memory and within each block sequentially process each tile. In contrast, in Mini Gunrock, we can only parallelize over all items globally,

which reduces the chance of collisions in hash tables. Another such difference is in pull-based traversal. In Mini Gunrock, when a thread finds its first parent node, it can only exit early from own kernel, but cannot do a group early exit of all the threads that are cooperatively working on that specific neighbor list. As a result, the performance gain of pull-based traversal in Mini Gunrock is limited compared to the same optimization implemented in Gunrock.

There are also several optimization implementations are similar in both Gunrock and Mini Gunrock, including the fusing of compute kernel and traversal operators and the two-level priority queue.

In general, any optimization that 1) only relies on input frontier and output frontier indices and 2) operates on one item only once can be implemented within the transform-based framework.

4.4 Graph Primitives using Transforms

One goal of Mini Gunrock is to quickly build new graph primitive prototypes with minimal code size. We discuss various graph primitives that can be implemented using Mini Gunrock.

Just as in Gunrock, the three main components of a graph primitive in Mini Gunrock are: 1) Problem, which provides graph topology data and an algorithm-specific data management interface; 2) Functor, which contains user-defined computation code as a device function; and 3) Enactor, which serves as the entry point of the graph primitive and defines the running process by a series of graph operators. To show how Mini Gunrock implements every primitive in Gunrock, we show the source code of the SSSP enactor and functor in Mini Gunrock (the SSSP problem code is very simple):

Listing 4.1: *Graph operator interfaces.*

```
template<typename Problem, typename Functor,
    bool idempotence, bool has_output>
int advance_forward_kernel(std::shared_ptr<
    Problem> problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
int iteration,
standard_context_t &context)

template<typename Problem, typename Functor>
int filter_kernel(std::shared_ptr<Problem>
    problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
int iteration,
standard_context_t &context)

template<typename Problem, typename Functor,
    typename Value, typename reduce_op, bool
    has_output>
int neighborhood_kernel(std::shared_ptr<
    Problem> problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
Value  *reduced,
Value identity,
int iteration,
standard_context_t &context)
```

In Mini Gunrock, we keep the operator interface to the minimum. As in Gunrock, each operator interface also contains its graph primitive data structure type (Problem), its functor structure (Functor), and some auxiliary variables. But for function arguments, Mini Gunrock only contains the problem object, the input and output frontier, the current iteration number, and a context object needed by all transform-based primitives.

Listing 4.2: *SSSP Enactor.*

```
void enact(std::shared_ptr<sssp_problem_t>
    sssp_problem, standard_context_t &context)
    {
        init_frontier(sssp_problem);
        int frontier_length = 1;
        int selector = 0;
        // Start the iteration with the initial
            frontier that contains the source
            node ID.
        for (int iteration = 0; ; ++iteration)
            {
            frontier_length =
                advance_forward_kernel<
                sssp_problem_t, sssp_functor_t,
                 false, true>
                 (sssp_problem, buffers[selector
                    ], buffers[selector^1],
                    iteration, context);
            selector ^= 1;
            frontier_length = filter_kernel<
                sssp_problem_t, sssp_functor_t>
```

```
                (sssp_problem, buffers[selector
                    ], buffers[selector^1],
                    iteration, context);
            // Use advance to relax node
                distance value and discover new
                 frontier, and filter to remove
                 redundant node IDs, until
                frontier is empty.
            if (!frontier_length) break;
            selector ^= 1;
        }
    }
```

Listing 4.3: *SSSP Functor.*

```
struct sssp_functor_t {
static __device__ __forceinline__ bool
    cond_filter(int idx, sssp_problem_t::
    data_slice_t *data, int iteration) {
    // If idx is not -1, then it's unique,
        return true to keep it.
    return idx != -1;
}
static __device__ __forceinline__ bool
    cond_advance(int src, int dst, int
    edge_id, int rank, int output_idx,
    sssp_problem_t::data_slice_t *data, int
    iteration) {
    // If source node's distance value plus
        the edge weight is larger than
        destination node's distance value,
        update destination node's distance
        value.
    float new_distance = data->d_labels[src]+
        data->d_weights[edge_id];
    float old_distance = atomicMin(data->
        d_labels+dst, new_distance);
    return (new_distance < old_distance);
}
static __device__ __forceinline__ bool
    apply_advance(int src, int dst, int
    edge_id, int rank, int output_idx,
    sssp_problem_t::data_slice_t *data, int
    iteration) {
    // If destination node's distance value
        is updated, set its predecessor to a
        new source node.
    data->d_preds[dst] = src;
    return true;
}
};
```

For SSSP, the enactor first initializes two buffers for input and output frontiers, and then calls a series of graph operators until it breaks when frontier is empty. The functor is similar to Gunrock and contains very simple user-defined computation code. Our design reduces the code size needed for both a new graph operator implementation and a new graph primitive using existing graph operators to more than 10 times smaller than Gunrock. The SSSP enactor and

functor only contains 208 lines of code (77 for enactor, 39 for functor, 92 for problem). Both BFS and PR implementation in Mini Gunrock are within 200 lines of code. The total lines of code of the advance, filter, and neighborhood reduction operator in Mini Gunrock are 47, 21, and 49 respectively.

4.4.1 Mini Gunrock vs. Gunrock

This section compares Mini Gunrock and Gunrock from the graph primitive perspective.

Mini Gunrock has implemented several graph primitives in Gunrock, such as BFS, SSSP, and graph coloring. It has also implemented one Gunrock graph primitive with different graph operators: PR. However, the current set of graph operators in Mini Gunrock cannot implement triangle counting in Gunrock.

The efficient implementation of neighborhood reduction in Mini Gunrock makes it capable of implementing several new graph primitives. Such examples include maximal independent set, weighted label propagation, and several other primitives whose main operator is neighborhood reduction. The flexibility of doing both in-degree and out-degree neighborhood reduction also enables us to implement node-ranking primitives with rank scores distributed along neighbors using either push-style or pull-style. Such examples include PageRank and betweenness centrality, where switching to a pull-style neighborhood reduction could avoid atomic operations and thus improve the overall performance. Because Mini Gunrock and Gunrock follow the same data-centric abstraction, the features that are missing from current Gunrock are mostly on the implementation side, and can be fixed in future Gunrock releases. For example, neighborhood reduction requires an additional segment reduction within our current load-balanced search implementation in Gunrock; in-degree and out-degree advance/neighborhood reduction requires some interface changes and additional runtime memory to store neighbor lists for both directions.

4.5 Performance Analysis

The set of datasets we used in our experiments is a subset of the datasets we used in chapter 6, except *kron*, which is an R-MAT graph with scale 21 and edge factor 48.

4.5.1 Performance Summary

Table 4.1 shows the end-to-end runtime comparison for BFS and PR of Mini Gunrock and Gunrock on five different datasets. For advance, we compare with Gunrock's baseline implementation, which uses the LB strategy and disables direction-optimizing traversal and idempotence since we have observed that: (1) idempotence will cause different per-iteration frontier sizes and make it difficult to compare Mini Gunrock and Gunrock; (2) direction-optimizing traversal has a wide performance gap between Mini Gunrock and Gunrock due to the limitation of pull-based traversal implementation in Mini Gunrock we have discussed in section 4.3.3. Based on these reasons, we think using baseline LB strategy in Gunrock is the most comparable setting to Mini Gunrock's transform-based implementation that also uses merge-based load balanced search.

The geomean speedup of Mini Gunrock over Gunrock for BFS and PR on three scale-free graphs (soc-orkut, hollywood, and kron) are 1.507 and 1.558 separately. This shows that both our transform-based graph operators in Mini Gunrock (the *lbs_transform*-based advance operator and the *segreduce_transform*-based neighborhood reduction operator) can achieve comparable performance with Gunrock's baseline advance operator on scale-free graphs. Why? In this comparison, LB advance in BFS and PR uses several separate GPU kernels for getting the length of neighbor lists in the input frontier, doing prefix-sum, doing sorted search, and mapping the workload to threads to cooperatively expand neighbor lists. In contrast, Mini Gunrock's transform-based implementation fuses these kernels into two transforms, significantly reducing data movement and increasing the per-iteration performance.

On the other hand, Mini Gunrock has an order-of-magnitude performance gap vs. Gunrock on BFS on graphs with large diameter and small average degree (roadnet and rgg). First, the transform-based operators have extra kernel launch overhead, so for graphs with large diameters,

Algorithm	Datasets	MGR Runtime (ms)	GR Runtime (ms)
BFS	soc-orkut	111.4	212.6
	hollywood	37.32	36.5
	kron	107.3	196.9
	roadnet	7548	622.6
	rgg	3473	485.7
PR	soc-orkut	88.28	176
	hollywood	27	27.31
	kron	93.91	176.2
	roadnet	365.9	91.81
	rgg	199.5	181.6

Table 4.1: *Runtime comparison of Mini Gunrock and Gunrock for BFS and PR. Note that for BFS, we force Gunrock to use the baseline LB strategy with no idempotence or direction-optimizing traversal, and for PR, we compare the runtime for one iteration.*

if the algorithm has a very large search depth, the accumulated overhead will significantly hurt the overall performance. Second, in Gunrock's implementation, graphs with a small average degree and a small size frontier cannot fully utilize the GPU. Thus Gunrock switches from load-balance on the output frontier (which will have several binary searches for neighbor lists visiting within one block) to load-balance on the input frontier. Since the average degree is small, after the switch, Gunrock's implementation will still have near-perfect load balancing and avoid the overhead of binary searches when the number of the small neighbor lists assigned to one block is far beyond the block dimension. However, Mini Gunrock's transform-based operator can only perform one strategy of load balancing, on the output frontier, and suffers a performance penalty as a result.

4.5.2 Per-Iteration Performance Analysis

To compare per-iteration operation performance between Mini Gunrock and Gunrock, we collect the runtime and MTEPS for each BFS advance operation on two scale-free graphs and two road network graphs. Without losing generality, we only compare the advance operator, which has the same workload and memory access pattern as neighborhood reduction operator, and ignore the filter operator which does not expose much irregularity.

Figure 4.5 shows the per-iteration advance operation MTEPS for both Mini Gunrock and Gunrock

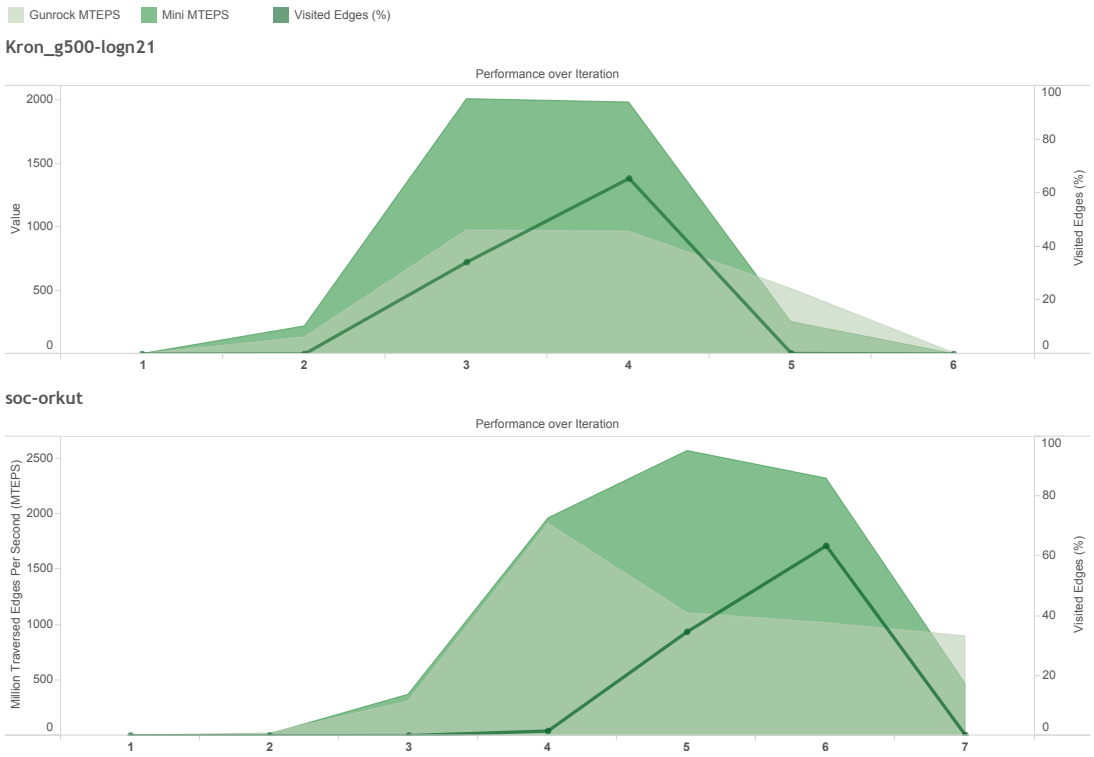


Figure 4.5: *Area chart that compares per-iteration advance performance between Gunrock and Mini Gunrock with the percentage of visited edges of that iteration.*

on two scale-free graphs. From the figure we can see that Mini Gunrock both shows better peak performance, and stays within the range of the peak performance for more iterations, than Gunrock. This demonstrates the efficiency of the transform-based approach.

Figure 4.6 shows the per-iteration advance operation MTEPS for both Mini Gunrock and Gunrock on two road network graphs. Neither shows peak performance due to the small size of frontiers. Both figures show Mini Gunrock's advance constantly performs worse than Gunrock when the frontier size is too small to fully utilize computing resources. The result also means that for small frontiers, both Gunrock and Mini Gunrock need to try alternative load balancing strategies, such as Gunrock's TWC strategy.

4.6 Limitations

The high-level abstraction and high performance of transform-based primitives allows several benefits when Mini Gunrock's graph operators use them without modification. However, it also

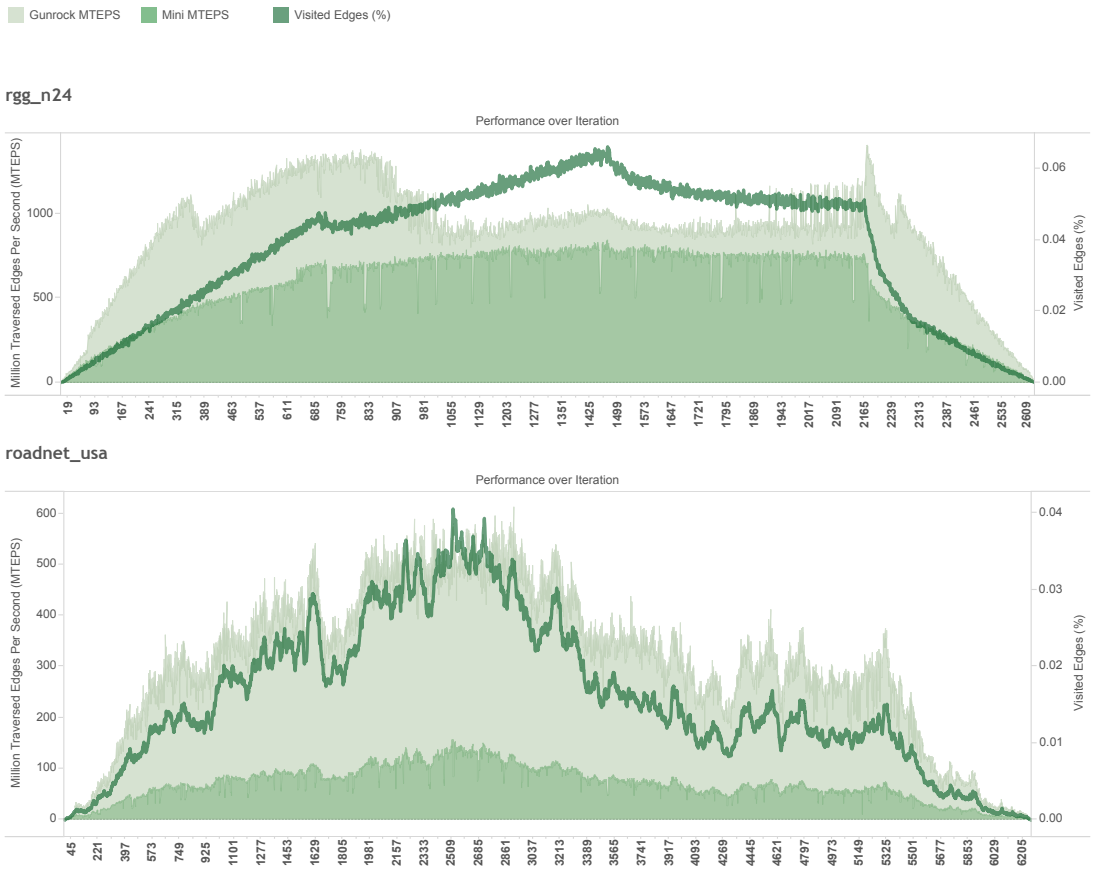


Figure 4.6: *Area chart that compares per-iteration advance performance between Gunrock and Mini Gunrock with the percentage of visited edges of that iteration.*

has some limitations if we want more flexibility. The two sources of these limitations are: 1) the transform abstraction and 2) the current transform implementation in moderngpu 2.0.

4.6.1 Limitations of the Transform Abstraction

Lack of Multi-Frontier Support: As we have noted in section 4.1, moderngpu supports only two types of transforms—unary transform and binary transform—which means that they cannot be applied to several multi-frontier required operators and optimizations in Gunrock. Such optimizations include multi-level priority queues based on multisplit [2] and integrated pull-and-push-based advance with both a sparse-compact current visiting frontier and a dense-bitmap unvisited frontier. Another such graph operator is segmented intersection, which requires two input frontiers. However, we expect that expanding the

current transform primitive design to support multiple frontiers would enable more potential optimizations and graph operators. One solution to go beyond the unary and binary transform limitation on the number of input and output array is to read from/write to additional arrays in global memory from the kernel function in the operator.

Kernel Launching Overhead: Compared to graph operators in Gunrock, graph operators built on top of transform-based primitives have more kernel launching overhead (see Table 4.1), which makes their performance suffer for graph datasets that have a large diameter. Using transform-based primitives also makes the kernel fusion of two graph operators impossible in Mini Gunrock.

4.6.2 Limitations of the Current Transform Implementation

Limited flexibility: The current implementation of moderngpu 2.0 automatically sets up an optimal launch setting and workload scheduling/decomposition phase according to a fixed number of items processed per thread. This can often find the most load-balanced and work-efficient workload mapping for existing primitives in moderngpu 2.0. However, customized optimizations such as a dynamic grouping workload mapping strategy [59] (DG) based on a persistent threads programming style are difficult to implement, as the current unary transform will apply a unary operation to every input item exactly once, but DG requires an input item to take charge of a thread group (either a warp or a block) to perform computation on neighbor lists, in which case each input item may process not only one unary operation more than once but also multiple unary operations. The opacity of the transform-based primitive's set-up and scheduling phase also increases the difficulty of implementing more complicated user-defined functions such as those that use shared memory, complex synchronization, and operations such as early exit on all threads within a block. An implementation with user-specific shared-memory and local-memory allocation capability as well as synchronization interfaces for synchronization among user-specified thread groups is the key feature to make the current transform implementation flexible enough for more optimizations in Gunrock.

Fixed Two-Phase Idiom: Moderngpu 2.0 breaks primitives into two distinct phases [4]: 1) find a coarse-grained partitioning of the problem that exactly load-balances work over each thread, and 2) execute simple, work-efficient, sequential logic that solves the problem. This two-phase idiom always searches for perfect load balance and does not allow user-specified coarse-grained partitioning strategies in the first phase. However, our experiments show that perfect load balance does not necessarily bring the best performance due to the potential overhead it will cause for the second execution phase. Thus a more flexible interface that allows users to try different strategies in both phases will be desirable for achieving better performance for different graph primitives on different datasets.

4.7 Conclusion

In this Chapter, we discuss the design and implementation of Mini Gunrock, a lightweight graph analytics framework on the GPU. It shows the flexibility of data-centric abstraction by implementing same abstraction with different implementation. We have also explored different performance tradeoffs, and identified strengths and weaknesses of transform-based abstraction and implementation. In conclusion, with Mini Gunrock, we have achieved our three goals of flexibility, simplicity, and comparable performance. It is able to have the same features for graph operators in Gunrock; it has enabled rapid prototyping of both new graph operators and new graph primitives, and it achieves comparable performance with Gunrock.

Chapter 5

Graph Applications

One of the principal advantages of Gunrock's abstraction is that our graph operators can be composed to build new graph primitives with minimal extra work. For each primitive below, we describe the specialized GPU implementation to which we compare, followed by how we express this primitive in Gunrock. We will compare the performance between specialized GPU implementation and Gunrock implementation in Chapter 6.

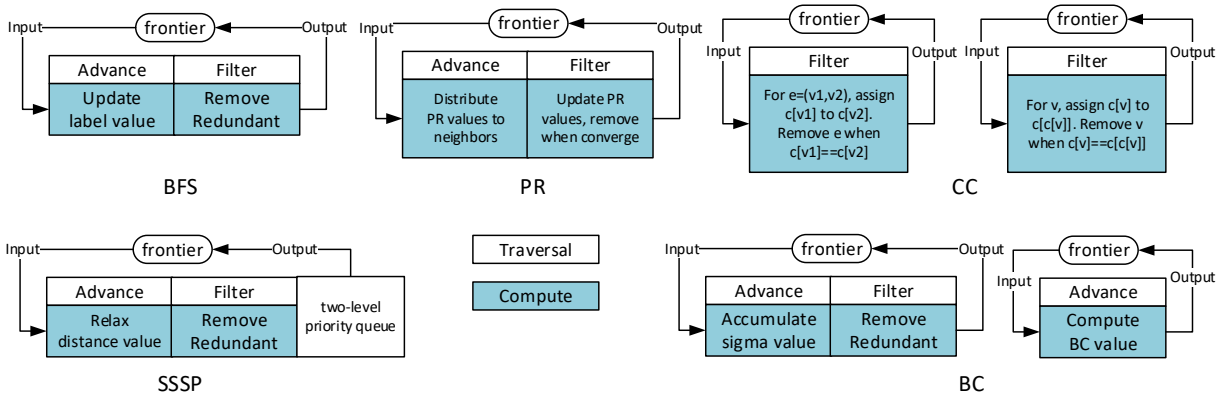


Figure 5.1: *Operation flow chart for selected primitives in Gunrock (two black lines with an arrow at one end indicates a while loop that runs until the frontier is empty).*

5.1 Traversal-based Primitives

One of the largest types of graph primitives are traversal-based primitives. This type of graph primitives often starts from a subset of graph (more often than not a single source vertex), then traverses the whole graph or a part of the graph by expanding the neighbor lists and doing computation over the corresponding edges or nodes. Gunrock supports several traversal-based primitives. We discuss BFS, SSSP, and how to use Gunrock for traversal-based primitives in general.

5.1.1 Breadth-First Search (BFS)

BFS initializes its vertex frontier with a single source vertex. On each iteration, it generates a new frontier of nodes with all unvisited neighbor nodes in the current frontier, setting their label values (hop distance from source node) and repeating until all nodes have been visited. BFS is one of the most fundamental graph primitives and serves as the basis of several other graph primitives when the label update is replaced with other per-node or per-edge computations.

Specialized GPU Implementation The well-known BFS implementation of Merrill et al. [59] achieves its high performance through careful load-balancing, avoidance of atomics, and heuristics for avoiding redundant vertex discovery. Its chief operations are expand (to generate a new frontier) and contract (to remove redundant nodes) phases.

Gunrock Implementation Merrill et al.'s expand maps nicely to Gunrock's advance operator, and contract to Gunrock's filter operator. During advance, we set a label value for each vertex to show the distance from the source, and/or set a predecessor value for each vertex that shows the predecessor vertex's ID. We implement efficient load-balancing (Section 3.2.1) and both push- and pull-based advance (Section 3.2.1.4) for more efficient traversal. Our base implementation uses atomics during advance to prevent concurrent vertex discovery. When a vertex is uniquely discovered, we set its label (depth) and/or predecessor ID. Gunrock's fastest BFS uses the idempotent advance operator (thus avoiding the cost of atomics) and uses heuristics within its filter that reduce the concurrent discovery of child

nodes (Section 3.2.2).

Algorithm 4 Breadth-First Search, expressed in Gunrock's abstraction

```
1: procedure Set_Problem_Data(G, P, root)
2:     P.labels[1..G.verts] ← ∞
3:     P.labels[root] ← 0
4:     P.frontier.Insert(root)
5: end procedure
6:
7: procedure UpdateLabel(s_id, d_id, e_id, P)
8:     return −1 == atomicCAS(P.labels[d_id], −1, new_label)
9: end procedure
10:
11: procedure RemoveRedundant(node_id, P)
12:     pick_node ← node_id ≠ −1
13:     return pick_node
14: end procedure
15:
16: procedure BFS_Enactor(G, P, root)
17:     Set_Problem_Data(G, P, root)
18:     while P.frontier.Size() > 0 do
19:         Advance(G, P, UpdateLabel)
20:         Filter(G, P, RemoveRedundant)
21:     end while
22: end procedure
```

Algorithm 4 shows the base line implementation in Gunrock without direction optimal, idempotence, and advance and filter fusion. With these optimization switches on, we are able to increase the performance of BFS even more in Gunrock while significantly reducing the code size users have to write.

5.1.2 Single-Source Shortest Path

Single-source shortest path finds paths between a given source vertex and all other nodes in the graph such that the weights on the path between source and destination nodes are minimized. While the advance mode of SSSP is identical to BFS, the computation mode differs. The most significant benefit that delta-stepping method for SSSP brings is the use of priority queue. It relaxes the one-vertex-at-a-time constraint imposed by a sequential Dijkstra's algorithm by grouping output frontier of each iteration into buckets and processing vertices in different buckets

in parallel. Gunrock's data-centric programming model is the only work on the GPU that supports such operation.

Specialized GPU Implementation Currently the highest performing SSSP algorithm implementation on the GPU is the work from Davidson et al. [18]. They provide two key optimizations in their SSSP implementation: 1) a load balanced graph traversal method and 2) a priority queue implementation that reorganizes the workload. Gunrock generalizes both optimization strategies into its implementation, allowing them to apply to other graph primitives as well as SSSP. We implement Gunrock's priority queue as an additional filter pass between two iterations.

Gunrock Implementation We start from a single source vertex in the frontier. To compute a distance value from the source vertex, we need one advance and one filter operator. On each iteration, we visit all associated edges in parallel for each vertex in the frontier and relax the distance's value (if necessary) of the nodes attached to those edges. We use an AtomicMin to atomically find the minimal distance value we want to keep and a bitmap flag array associated with the frontier to remove redundant nodes. After each iteration, we use a priority queue to reorganize the nodes in the frontier. Not as the original delta-stepping SSSP work [60], our priority queue operator only contains two levels which can be generated efficiently using two passes of stream compaction process. The algorithm is shown in algorithm 1.

5.1.3 Extension to Other Traversal-Based Primitives

Using Gunrock's data-centric programming model, implementing new traversal-based primitives is simple. One proof is the enactor of BFS and SSSP are the same, by only modifying the functor in compute, we can simply build SSSP from BFS. Using the same strategy, we can use Gunrock to build traversal-based primitives from maxflow to any tree-based search algorithm.

5.2 Ranking Primitives

Another important type of graph primitives is ranking primitives. This type of graph primitives use different metrics to compute a rank value for each node or edge, the ranking scores are then used to reveal relative importantness of nodes and edges in the graph. Gunrock supports several ranking primitives. We discuss BC, Pagerank, and three node ranking primitives on bipartite graph: HITS, SALSA, and Twitter's Who-to-follow.

5.2.1 Betweenness Centrality

The BC index can be used in social network analysis as an indicator of the relative importance of nodes in a graph. At a high level, the BC for a vertex in a graph is the fraction of shortest paths in a graph that pass through that vertex. A näive method is to perform a BFS starting at each vertex to compute the pair-dependencies, and then sum the pair -dependencies for each $v \in V$, which results in a $O(|V|^3)$ complexity. Brandes's BC formulation [10] computes the dependency of the source node s on all vertices v recursively:

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su} \tag{5.1}$$

$$\delta_{s\bullet}(v) = \sum_{w:v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \times (1 + \delta_{s\bullet}(w)) \tag{5.2}$$

These two equations are most commonly used for GPU implementations due to its $O(|V||E| + |V|^2)$ complexity by avoiding the explicit summation of pair-dependencies.

Specialized GPU Implementation Brandes's formulation has two passes: a forward BFS pass to accumulate sigma values for each node, and a backward BFS pass to compute centrality values. Jia et al. [39] and Sariyüce et al. [72] both use an edge-parallel method to implement the above two passes. We achieve this in Gunrock using two advance operators on an edge frontier with different computations. The recent (specialized) multi-GPU BC algorithm by McLaughlin and Bader [57] uses task parallelism, dynamic load balancing, and sampling

techniques to perform BC computation in parallel from different sources on different GPU SMXs.

Gunrock Implementation Gunrock's implementation also contains two phases. The first phase has an advance step identical to the original BFS and a computation step that computes the number of shortest paths from source to each vertex. The second phase uses an advance step to iterate over the BFS frontier backwards with a computation step to compute the dependency scores. We achieve competitive performance on scale-free graphs with the latest hardwired BC algorithm [56]. Within Gunrock, we haven't yet considered task parallelism since it appears to be limited to BC, but it is an interesting area for future work. We can generate the right frontier queue for each iteration in the backward phase using push-based traversal and a filter operator or pull-based traversal with pre-stored frontiers during the forward BFS phase with extra memory cost.

Algorithm 5 Betweenness Centrality, expressed in Gunrock's abstraction

```
1: procedure Set_Problem_Data(G, P, root)
2:     P.sigmas[1..G.verts] ← 0
3:     P.sigmas[root] ← 1
4:     P.labels[1..G.verts] ← ∞
5:     P.labels[root] ← 0
6:     P.deltas[1..G.verts] ← 0
7:     P.frontier.Insert(root)
8: end procedure
9: procedure UpdateSigmas(s_id, d_id, e_id, P)
10:    atomicAdd(P.sigmas[d_id], P.sigmas[s_id])
11:    return −1 == atomicCAS(P.labels[d_id], −1, new_label)
12: end procedure
13: procedure RemoveRedundant(node_id, P)
14:    pick_node ← node_id ≠ −1
15:    return pick_node
16: end procedure
17: procedure SelectDstNodes(s_id, d_id, e_id, P)
18:    return P.labels[d_id] == P.labels[s_id] + 1
19: end procedure
20: procedure ComputeBC(s_id, d_id, e_id, P)
21:    result ← P.sigmas[s_id]/P.sigmas[d_id] ∗ (1.0 + P.deltas[d_id])
22:    atomicAdd(P.deltas[s_id], result)
23: end procedure
24: procedure SelectSourceNodes(node_id, iteration, P)
25:    return P.labels[node_id] == iteration
26: end procedure
27: procedure BC_Enactor(G, P, root) Set_Problem_Data(G, P, root)
28:    while P.frontier.Size() > 0 do
29:        Advance(G, P, UpdateLabel)
30:        Filter(G, P, RemoveRedundant)
31:    end while
```

```
32:    iteration ← max_traversal_depth − 1
33:    while iteration > 0 do
34:        iteration ← iteration − 1
35:        Filter(G, P, SelectSourceNodes)
36:        Advance(G, P, SelectDstNodes, ComputeBC)
37:    end while
38: end procedure
```

5.2.2 PageRank and Other Node Ranking Algorithms

PageRank is an algorithm for ranking nodes in a graph based on the structure of the graph [63]. It was originally used on the web graph to rank webpages for search engines. The PageRank of a node can be thought of as the probability that a random walker traversing the graph along the edges will land on that node. It is a measure of how well-connected the node is, and therefore, how important it is to the graph. The iterative method of computing PageRank gives each vertex an initial PageRank value and updates it based on the PageRank of its neighbors, until the PageRank value for each vertex converges. PageRank is one of the simplest graph algorithms to implement on GPUs because the frontier always contains all nodes, so its computation is congruent to sparse matrix-vector multiply; because it is simple, most GPU frameworks implement it in a similar way and attain similar performance.

In Gunrock, we begin with a frontier that contains all nodes in the graph and end when all nodes have converged. Each iteration contains one advance operator to compute the PageRank value on the frontier of nodes, and one filter operator to remove the nodes whose Pageranks have already converged. We accumulate PageRank values with AtomicAdd operations. The PageRank primitive could further be optimized with a neighborhood reduction operator with all-node input frontier awareness so that the load balancing only need to be done once. A Personalized PageRank (PPR) calculation relative to node N is identical to the normal PageRank calculation, except all random walks begin at node N, rather than a random node. Overall, a Personalized PageRank calculation for N shows which nodes are most closely related to A. We will discuss it in section 5.2.2.1.

Algorithm 6 PageRank, expressed in Gunrock's abstraction

```
1: procedure Set_Problem_Data(G, P, delta)
2:     P.cur_rank[1..G.verts] ← 1 − delta
3:     P.next_rank[1..G.verts] ← 1 − delta
4:     P.frontier.Insert(P.V)
5: end procedure
6:
7: procedure DistributeRank(s_id, d_id, e_id, P)
8:     atomicAdd(P.next_rank[d_id], P.cur_rank[s_id]/P.degree[s_id])
9: end procedure
10:
11: procedure UpdateRank(node_id, P)
12:     P.next_rank[node_id] ← (1.0 − delta) + delta ∗ P.next_rank[node_id]
13:     diff ← fabs(P.cur_rank[node_id], P.next_rank[node_id])
14:     return (diff > error_threshold)
15: end procedure
16:
17: procedure PR_Enactor(G, P, delta, error_threshold, max_iter)
18:     Set_Problem_Data(G, P, delta)
19:     iteration ← 0
20:     while P.frontier.Size() > 0||iteration < max_iter do
21:         iteration ← iteration + 1 Advance(G, P, DistributeRank)
22:         Filter(G, P, UpdateRank)
23:         SwapRankPointers(P.cur_rank, P.next_rank)
24:     end while
25: end procedure
```

5.2.2.1 Node Ranking Algorithms on Bipartite Graphs

Several node ranking algorithms run on bipartite graphs. We implement bipartite graphs in Gunrock in the following way. A directed bipartite graph is similar to a normal undirected graph, except in a directed bipartite graph, we need to consider the outgoing edges and the incoming edges of a vertex separately. We achieve this by reversing the source and destination vertex ID for each edge while constructing the CSR data structure to record the incoming edges and using an additional prefix sum pass to compute the incoming degree for each vertex.

We use Gunrock to implement Twitter's who-to-follow algorithm [22] ("Money" [24]), which incorporated three node-ranking algorithms based on bipartite graphs (Personalized PageRank, Stochastic Approach for Link-Structure Analysis (SALSA) [48], and Hyperlink-Induced Topic

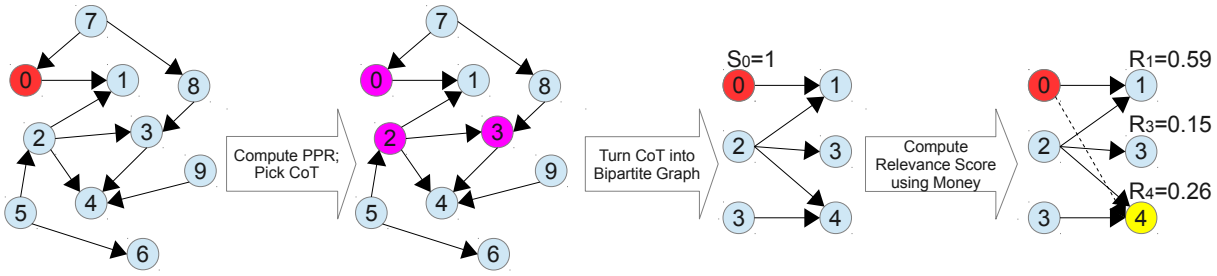


Figure 5.2: *Overview of Twitter's WTF algorithm.* Frame 1: *The initial graph (red [dark] node is the user for whom recommendations are being computed).* Frame 2: *The Circle of Trust (nodes in pink [dark]) is found using Personalized PageRank.* Frame 3: *The graph is pruned to include only the CoT and the users they follow.* Frame 4: *The relevance scores of all users on the right side are computed with Twitter's Money algorithm. Node 4 will be suggested for Node 0 to follow because it has the highest value of all nodes the user is not already following.*

Search (HITS) [44]). Two main stages comprise the WTF recommender. In the first stage, Twitter calculates a Personalized PageRank (PPR) for the user. PPR assigns a ranking to all nodes in the network based on how closely connected they are to the main node (the user interested in recommendations). This ranking is used to find the top 1000 ranking nodes. These nodes form the user's "Circle of Trust" (CoT), which consists of the 1000 nodes closest to the user. Pruning the graph in this way increases the personalization of the recommendation and reduces spam. Next, they create a bipartite graph with the CoT on one side, and the users followed by the CoT on the other. All other nodes are pruned from the graph. The final step is Twitter's "Money" algorithm, a graph analysis algorithm that determines which accounts the user is most likely to be interested in following. Figure 5.2 shows a schematic of the entire WTF algorithm.

Our implementation, the first to use a programmable framework for bipartite graphs, demonstrated that Gunrock's advance operator is flexible enough to encompass all three node-ranking algorithms, including a 2-hop traversal in a bipartite graph and a switch to choose between visiting the outgoing edges of a vertex and visiting the incoming edges of a vertex. We use this switch in our SALSA implementations. In our Money and HITS implementation, we do not need to calculate the in-degree of every node in the graph, but just the ones connected to the CoT. We take advantage of this by finding the incoming degree values for neighbors of the CoT on the fly with an additional pass of advance. Because of the small size of the CoT, this extra pass takes negligible time but saves us gigabytes of memory for large datasets.

A high level description is as follows: we first compute the PPR value for each vertex in the follow graph with a user-defined seed vertex. Then we radix-sort the PPR values and take the vertices with the top k PPR values (in our implementation, $k = 1000$) as the CoT and put them in a frontier. We then run the Money algorithm, sort the vertices that the CoT follows, and finally extract the vertices with the top relevance scores to use for our recommendation.

PPR: Algorithm 7 shows our implementation of PPR using Gunrock. We update PPR using the following equation:

$$PPR(v_i) = \begin{cases} (1-\delta) + \delta \cdot \sum\limits_{v_j \in pred(v_i)} \frac{PPR(v_j)}{d_{\text{OUT}}(v_j)} & v_i \text{ is seed} \\ \delta \cdot \sum\limits_{v_j \in pred(v_i)} \frac{PPR(v_j)}{d_{\text{OUT}}(v_j)} & \text{otherwise} \end{cases} \tag{5.3}$$

where δ is a constant damping factor (typically set to 0.85), $d_{\text{OUT}}(v_j)$ is the number of outbound edges on vertex v_j, and N is the total number of vertices in the graph.

The PPR algorithm starts by initializing problem data (line 1). It assigns the initial rank value for each vertex as $\frac{1}{N}$, and puts all vertices in the initial frontier. The main loop of the algorithm contains two steps. For each iteration, we first use advance to visit all the edges for each vertex in the frontier and distribute each vertex's PPR value to its neighbors (line 7). Then we use filter to update the PPR value for the vertices that still have unconverged PPR values. We give the seed vertex an extra value (line 10) as in the equation. Then we remove the vertices whose PPR values have converged from the frontier (line 12). The algorithm ends when all the PPR values converge and the frontier is empty.

Algorithm 7 Personalized PageRank

1: **procedure** Set_Problem_Data($G, P, seed, delta$)
2: $P.ranks\_curr[1..G.verts] \leftarrow \frac{1}{N}$
3: $P.src \leftarrow seed, P.delta \leftarrow delta$
4: $P.frontier.Insert(G.nodes)$
5: **end procedure**
6: **procedure** DistributePPRValue(s_id, d_id, P)
7: $atomicAdd(P.rank\_next[d\_id], \frac{P.rank\_curr[s\_id]}{P.out\_degree[s\_id]})$
8: **end procedure**
9: **procedure** UpdatePPRValue($node\_id, P$)
10: $P.rank\_next[node\_id] \leftarrow (P.delta \cdot P.rank\_next[node\_id]) + (P.src == node\_id)?(1.0 - P.delta) : 0$
11: $diff \leftarrow fabs(P.rank\_next[node\_id] - P.rank\_curr[node\_id])$
12: **return** $diff > P.threshold$
13: **end procedure**
14: **procedure** Compute_PPR($G, P, seed, delta, max\_iter$)
15: Set_Problem_Data($G, P, seed, delta$)
16: **while** $P.frontier.Size() > 0$ **do**
17: Advance($G, P, DistributePPRValue$)
18: Filter($G, P, UpdatePPRValue$)
19: $swap(P.rank\_next, P.rank\_curr)$
20: **end while**
21: **end procedure**

Money Algorithm: Algorithm 8 shows our implementation of Twitter's Money algorithm. We treat people in the CoT and people they follow as two disjoint sets X and Y in the bipartite graph $G = (X \cup Y, E)$ they form. For each node in X (the CoT), the Money algorithm computes its similarity score; for each node in Y, the Money algorithm computes its relevance score. We use the following equations in our implementation:

$$sim(x) = \begin{cases} \alpha + (1-\alpha) \cdot \sum\limits_{(x,y) \in E} \frac{relevance(y)}{d_{\text{IN}}(y)} & x \text{ is seed} \\ (1-\alpha) \cdot \sum\limits_{(x,y) \in E} \frac{relevance(y)}{d_{\text{IN}}(y)} & \text{otherwise} \end{cases}$$

$$relevance(y) = \sum_{(x,y) \in E} \frac{sim(x)}{d_{\text{OUT}}(x)}$$

In the Money algorithm, we also initialize problem data first (line 1). We set the similarity score for the seed vertex to 1, and set the similarity and relevance scores for all other

vertices to 0. We then put all vertices in the CoT in the initial frontier. The main loop of the algorithm contains two steps of advance, for each iteration we use the first advance to visit all the edges for each vertex in the CoT and distribute each vertex's similarity score to its neighbors' relevance scores (line 9). After this step, we update the relevance scores so that they may be used in the computation of similarity scores (line 19). The second advance will visit all the edges for each vertex in the CoT again, but this time it will distribute the neighbors' relevance scores back to each CoT vertex's similarity score (line 12). Again, we treat the seed vertex differently so that it can get more similarity score during each iteration, which in turn will affect its neighbors' relevance scores and other CoT vertices' similarity scores. After the second advance step, we update the similarity scores (line 21). As Twitter does in their Money algorithm [24], we end our main loop after $1/\alpha$ iterations.

Algorithm 8 Twitter's Money Algorithm

1: **procedure** Set_Problem_Data($G, P, seed, alpha$)
2: $P.relevance\_curr[1..G.verts] \leftarrow 0$
3: $P.sim\_curr[1..G.verts] \leftarrow 0$
4: $P.src \leftarrow seed, P.alpha \leftarrow alpha$
5: $P.sim\_curr[seed] \leftarrow 1$
6: $P.frontier.Insert(G.cot\_queue)$
7: **end procedure**
8: **procedure** DistributeRelevance(s_id, d_id, P)
9: $atomicAdd(P.relevance\_next[d\_id], \frac{P.sim\_curr[s\_id]}{P.out\_degree[s\_id]})$
10: **end procedure**
11: **procedure** DistributeSim(s_id, d_id, P)
12: $val \leftarrow (1 - P.alpha) \cdot \frac{P.relevance\_curr[d\_id]}{P.in\_degree[d\_id]} + (P.src == s\_id)?(\frac{P.alpha}{P.out\_degree[s\_id]}) : 0$
13: $atomicAdd(P.sim\_next[d\_id], val)$
14: **end procedure**
15: **procedure** Compute_Money($G, P, seed, alpha$)
16: Set_Problem_Data($G, P, seed, alpha$)
17: **for** $iter^{++} < 1/alpha$ **do**
18: Advance($G, P, DistributeRelevance$)
19: $swap(P.relevance\_next, P.relevance\_curr)$
20: Advance($G, P, DistributeSim$)
21: $swap(P.sim\_next, P.sim\_curr)$
22: **end for**
23: **end procedure**

5.3 Clustering Primitives

Clustering primitives group the vertices of the graph into clusters taking into consideration the edge structure of the graph in such a way that there should be many edges within each cluster and relatively few between the clusters. When we allow no edge between clusters, the clustering primitive is equal to connected component labeling primitive. In this dissertation, we only focus on clustering primitives that are based on edge-based agglomerative/divisive methods, which often use label propagation and neighborhood reduction operators.

5.3.1 Connected Component Labeling

The connected component primitive is a special clustering primitive that labels the nodes in each connected component in a graph with a unique component ID.

Specialized GPU Implementation Soman et al. [81] base their implementation on two PRAM algorithms: hooking and pointer-jumping. Hooking takes an edge as the input and tries to set the component IDs of the two end nodes of that edge to the same value. In odd-numbered iterations, the lower vertex writes its value to the higher vertex, and vice versa in the even numbered iteration. This strategy increases the rate of convergence. Pointer-jumping reduces a multi-level tree in the graph to a one-level tree (star). By repeating these two operators until no component ID changes for any node in the graph, the algorithm will compute the number of connected components for the graph and the connected component to which each node belongs.

Gunrock Implementation Gunrock uses a filter operator on an edge frontier to implement hooking. The frontier starts with all edges and during each iteration, one end vertex of each edge in the frontier tries to assign its component ID to the other vertex, and the filter step removes the edge whose two end nodes have the same component ID. We repeat hooking until no vertex's component ID changes and then proceed to pointer-jumping, where a filter operator on nodes assigns the component ID of each vertex to its parent's component ID until it reaches the root. Then a filter step removes the node whose component ID equals

its own node ID. The pointer-jumping phase also ends when no vertex's component ID changes.

Algorithm 9 Connected Component Labeling, expressed in Gunrock's abstraction

```
1: procedure Set_Problem_Data(G, P, delta)
2:     P.mark[1..G.edges] ← 1
3:     P.component_id[1..G.verts] ← (1..G.verts)
4:     P.edge_flag ← false
5:     P.node_flag ← false
6:     P.frontier.Insert(P.E)
7: end procedure
8:
9: procedure Hook(e_id, P, is_max_hook)
10:    if P.mark[e_id] then
11:        src_id ← P.src_id[e_id]
12:        dst_id ← P.dst_id[e_id]
13:        src_parent ← P.component_id[src_id]
14:        dst_parent ← P.component_id[dst_id]
15:        max_node ← P.max(src_parent, dst_parent)
16:        min_node ← src_parent + dst_parent − max_node
17:        if max_node == min_node then
18:            P.mark[e_id] ← true
19:        else
20:            if is_max_hook then
21:                P.component_id[max_node] ← min_node
22:            else
23:                P.component_id[min_node] ← max_node
24:            end if
25:            P.edge_flag ← false
26:        end if
27:    end if
28: end procedure
```

5.4 Global Indicator Primitives

Global indicator primitives are a type of graph primitives that conduct the computation and output a single or a series indicator value that shows some useful property of the graph. Simple regular tasks include computing degree distribution, producing top-K degree centrality nodes, more complicated tasks include using multi-BFS to estimate graph radii, and computing clustering coefficient value using triangle counting.

```
29: procedure PointerJump(node, P)
30:     parent ← P.component_id[node]
31:     grand_parent ← P.component_id[parent]
32:     if parent ≠ grand_parent then
33:         P.mark[e_id] ← true
34:         P.node_flag ← false
35:         P.component_id[node] ← grand_parent
36:     end if
37: end procedure
38:
39: procedure CC_Enactor(G, P)
40:     Set_Problem_Data(G, P)
41:     while P.edge_flag == false do
42:         P.edge_flag ← true
43:         Filter(G, P, Hook)
44:     end while
45:     P.frontier.Insert(P.V)
46:     while P.node_flag == false do
47:         P.node_flag ← true
48:         Filter(G, P, PointerJump)
49:     end while
50: end procedure
```

5.4.1 Triangle Counting

The extensive survey by Schank and Wagner [73] shows several sequential algorithms for counting and listing triangles in undirected graphs. Two of the best performing algorithms, *edge-iterator* and *forward*, both use edge-based set intersection primitives. The optimal theoretical bound of this operation coupled with its high potential for parallel implementation make this method a good candidate for GPU implementation.

We view the TC problem as a set intersection problem [87] by the following observation: An edge $e = (u, v)$, where u, v are its two end nodes, can form triangles with edges connected to both u and v. Let the intersections between the neighbor lists of u and v be $(w_1, w_2, \ldots, w_N)$, where N is the number of intersections. Then the number of triangles formed with e is N, where the three edges of each triangle are $(u, v), (w_i, u), (w_i, v), i \in [1, N]$. In practice, computing intersections for every edge in an undirected graph is redundant. We visit all the neighbor lists using Gunrock's *Advance* operator. If two nodes u and v have two edges (u, v) and (v, u) between them, we only

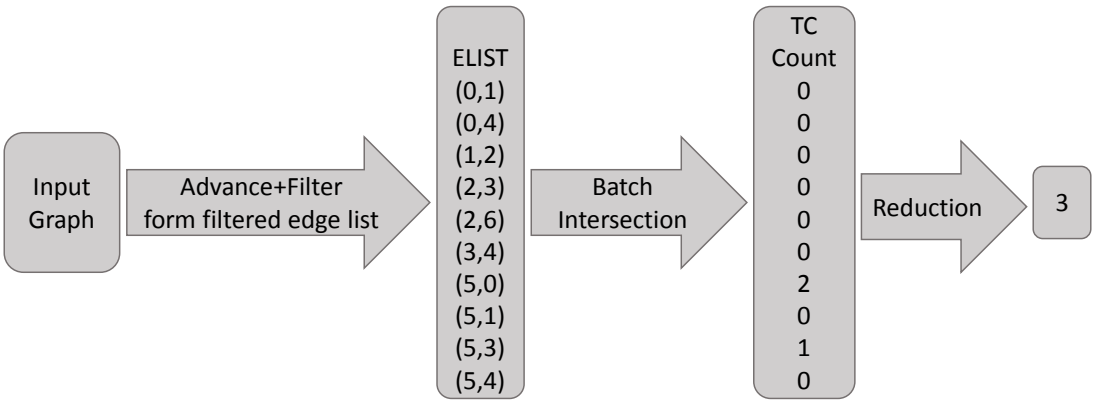


Figure 5.3: *The workflow of intersection-based GPU TC algorithm.*

keep one edge that points from the node with larger degree to the node with smaller degree. This will essentially reduce the number of the edges to process to half. Thus, in general, set intersection-based TC algorithms have two stages: (1) forming edge lists; (2) computing set intersections for two neighbor lists of an edge. Different optimizations can be applied to either stage. Our GPU implementation (shown in Algorithm 10) follows the *forward* algorithm and uses advance, filter, as well as segmented intersection operators in Gunrock. Figure 5.3 is the flow chart that shows how this algorithm works on the running example (Figure 3.1).

Algorithm 10 TC using edge-based set intersection.

```
1: procedure Form_Filtered_Edge_List(G)
2:     Advance(G)
3:     Filter(G)
4:     return ELIST
5: end procedure
6:
7: procedure Compute_Intersection(G, ELIST)
8:     {SmallList,LargeList} = Partition(G, ELIST)
9:     LargeNeighborListIntersection(LargeList)
10:    SmallNeighborListIntersection(SmallList)
11: end procedure
12:
13: procedure TC(G)
14:    ELIST=Form_Filtered_Edge_List(G)
15:    IntersectList=Compute_Intersection(G, ELIST)
16:    Count=Reduce(IntersectList)
17:    return Count
18: end procedure
```

5.5 Other Applications

Graph primitives we have implemented in Gunrock show that the data-centric programming model is expressive enough to cover various types of graph primitives and different ways to conduct graph traversal and computations. Beyond the graph primitives we discuss in this section, we have developed or are actively developing several other graph primitives in Gunrock using our data-centric programming model, including minimal spanning tree, maximal independent set, graph coloring, weighted label propagation, and subgraph matching.

Chapter 6

Performance Characterization

One important goal of Gunrock is to achieve comparable performance with specialized GPU graph primitives using a high-level abstraction. We show this by giving detailed performance characterization on Gunrock's five representative graph primitives: BFS, SSSP, BC, CC, and PR . We also provide micro-benchmarks to measure the impact of each individual optimization strategy on the performance, as well as how dataset-related attributes, such as degree distribution and frontier size, affect the performance. Finally, we provide performance analysis on two more complex graph primitives in Gunrock: TC and Who-to-Follow.

6.1 Overall Performance Analysis

We first show overall performance analysis of Gunrock on nine datasets including both real-world and generated graphs; the topology of these datasets spans from regular to scale-free.

We summarize the datasets in table 6.1. Soc-orkut (soc-ork), Soc-Liverjournal1 (soc-lj), and hollywood-09 (h09) are three social graphs; indochina-04 (i04) is a crawled hyperlink graph from indochina web domains; rmat_s22_e64 (rmat-22), rmat_s23_e32 (rmat-23), and rmat_s24_e16 (rmat-24) are three generated R-MAT graphs. All seven datasets are scale-free graphs with diameters of less than 30 and unevenly distributed node degrees (80% of nodes have degree less than 64). Both rgg_n_24 (rgg) and roadnet_USA (roadnet) datasets have large diameters with

Dataset	Vertices	Edges	Max Degree	Diameter	Type
soc-orkut	3M	212.7M	27,466	9	rs
soc-Livejournal1	4.8M	85.7M	20,333	16	rs
hollywood-09	1.1M	112.8M	11,467	11	rs
indochina-04	7.4M	302M	256,425	26	rs
rmat_s22_e64	4.2M	483M	421,607	5	gs
rmat_s23_e32	8.4M	505.6M	440,396	6	gs
rmat_s24_e16	16.8M	519.7M	432,152	6	gs
rgg_n_24	16.8M	265.1M	40	2622	gm
roadnet_USA	23.9M	577.1M	9	6809	rm

Table 6.1: *Dataset Description Table. Graph types are: r: real-world, g: generated, s: scale-free, and m: mesh-like. All datasets have been converted to undirected graphs. Self-loops and duplicated edges are removed.*

Algorithm	Galois	BGL	PowerGraph	Medusa
BFS	8.812	—	—	22.49
SSSP	2.532	99.99	8.058	*2.158*
BC	1.57	32.12	—	—
PageRank	2.16	—	17.73	*2.463*
CC	1.745	341.1	182.7	—

Table 6.2: *Geometric-mean runtime speedups of Gunrock on the datasets from Table 6.1 over frameworks not in Table 6.3. Due to Medusa's memory limitations [92], its SSSP and PageRank comparisons were measured on four smaller datasets.*

small and evenly distributed node degrees (most nodes have degree less than 12). soc-ork is from Network Repository; soc-lj, i04, h09, and roadnet are from UF Sparse Matrix Collection; rmat-22, rmat-23, rmat-24, and rgg are R-MAT and random geometric graphs we generated. The edge weight values (used in SSSP) for each dataset are random values between 1 and 64.

Performance Summary Tables 6.2 and 6.3, and Figure 6.1 and Figure 6.2, compare Gunrock's performance against several other graph libraries and hardwired GPU implementations.

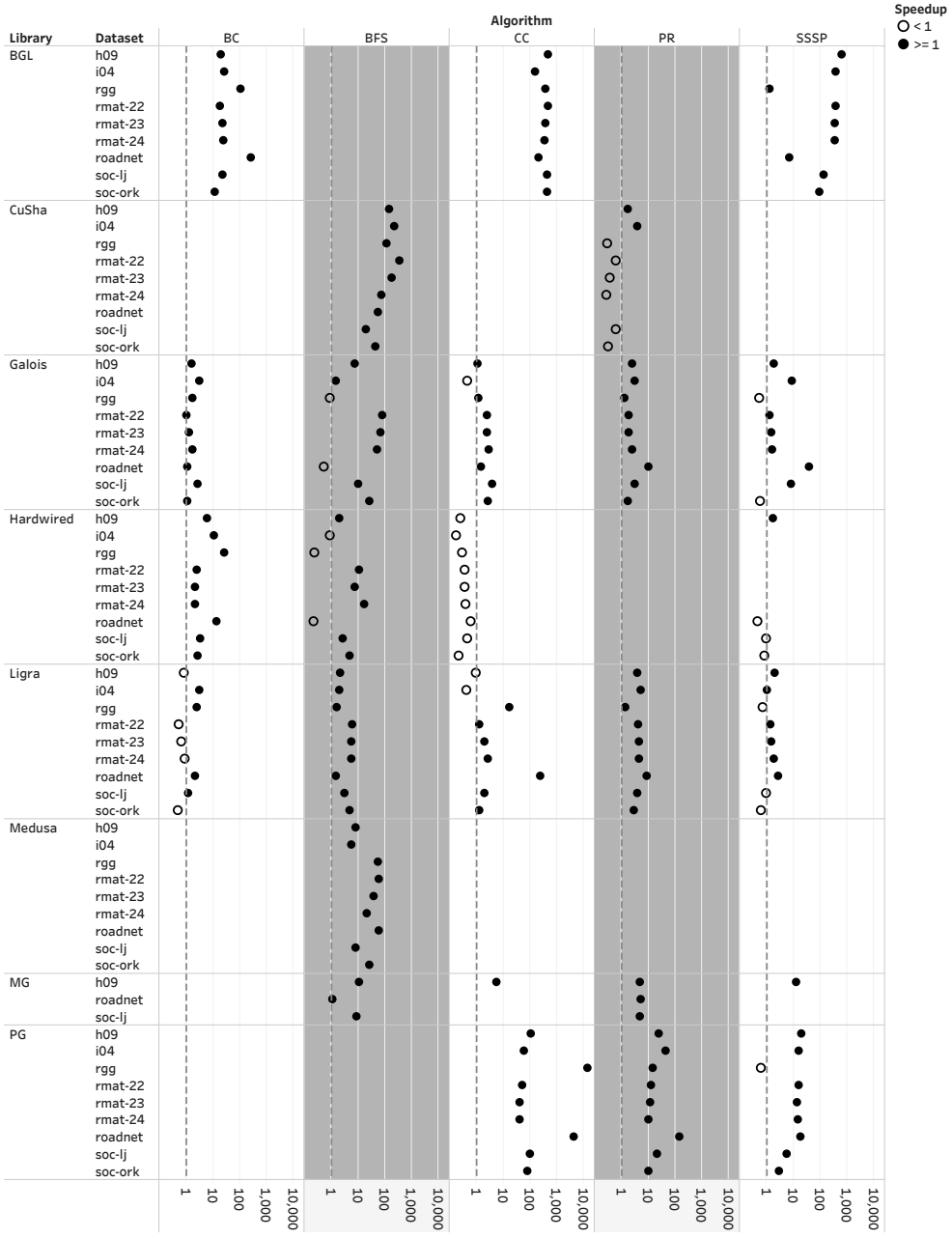


Figure 6.1: *Execution-time speedup for Gunrock vs. five other graph processing libraries/hard-wired algorithms on nine different graph inputs. Data is from Table 6.3. Black dots indicate Gunrock is faster, white dots slower.*

		Runtime (ms) [lower is better]					Edge throughput (MTEPS) [higher is better]				
Alg.	Dataset	CuSha	MapGraph	Hardwired GPU	Ligra	Gunrock	CuSha	MapGraph	Hardwired GPU	Ligra	Gunrock
BFS	soc-ork	244.9	OOM	25.81	26.1	5.573	868.3	OOM	12360	8149	38165
	soc-lj	263.6	116.5	36.29	42.4	14.05	519.5	1176	5661	2021	6097
	h09	855.2	63.77	11.37	12.8	5.835	131.8	1766	14866	8798	19299
	i04	17609	OOM	67.7	157	77.21	22.45	OOM	8491	1899	3861
	rmat-22	1354	OOM	41.81	22.6	3.943	369.1	OOM	17930	21374	122516
	rmat-23	1423	OOM	59.71	45.6	7.997	362.7	OOM	12971	11089	63227
	rmat-24	1234	OOM	270.6	89.6	16.74	426.4	OOM	—	5800	31042
	rgg	68202	OOM	138.6	918	593.9	3.887	OOM	2868	288.8	466.4
	roadnet	36194	763.4	141	978	676.2	3.189	151	1228	59.01	85.34
SSSP	soc-ork	—	OOM	807.2	595	981.6	—	OOM	770.6	—	216.7
	soc-lj	—	—	369	368	393.2	—	—	1039	—	217.9
	h09	—	1069	143.8	164	83.2	—	—	1427	—	1354
	i04	—	OOM	—	397	371.8	—	OOM	—	—	801.7
	rmat-22	—	OOM	—	774	583.9	—	OOM	—	—	827.3
	rmat-23	—	OOM	—	1110	739.1	—	OOM	—	—	684.1
	rmat-24	—	OOM	—	1560	884.5	—	OOM	—	—	587.5
	rgg	—	OOM	—	80800	115554	—	OOM	—	—	2.294
	roadnet	—	OOM	4860	29200	11037	—	OOM	25.87	—	5.229
BC	soc-ork	—	—	1029	186	397.8	—	—	413.3	4574	1069
	soc-lj	—	—	492.8	180	152.7	—	—	347.7	1904	1122
	h09	—	—	441.3	59	73.36	—	—	510.3	7635	3070
	i04	—	—	1270	362	117	—	—	469	3294	5096
	rmat-22	—	—	1867	399	742.6	—	—	517.5	4840	1301
	rmat-23	—	—	2102	646	964.4	—	—	481.3	3130	1049
	rmat-24	—	—	2415	978	1153	—	—	430.3	2124	901.2
	rgg	—	—	26938	2510	1023	—	—	19.69	422.5	518.4
	roadnet	—	—	15803	2490	1204	—	—	7.303	92.7	95.85
PageRank	soc-ork	52.54	OOM	—	476	173.1					
	soc-lj	33.61	250.7	—	200	54.1					
	h09	34.71	93.48	—	77.4	20.05					
	i04	164.6	OOM	—	210	41.59					
	rmat-22	188.5	OOM	—	1250	304.5					
	rmat-23	147	OOM	—	1770	397.2					
	rmat-24	128	OOM	—	2180	493.2					
	rgg	53.93	OOM	—	247	181.3					
	roadnet	—	123.2	—	209	24.11					
CC	soc-ork	—	OOM	46.97	260	211.7					
	soc-lj	—	OOM	43.51	184	93.27					
	h09	—	547.1	24.63	90.8	96.15					
	i04	—	OOM	130.3	315	773.7					
	rmat-22	—	OOM	149.4	563	429.8					
	rmat-23	—	OOM	212	1140	574.3					
	rmat-24	—	OOM	256.7	1730	664.1					
	rgg	—	OOM	103.9	6000	355.2					
	roadnet	—	OOM	124.9	50500	208.9					

Table 6.3: *Gunrock's performance comparison (runtime and edge throughput) with other graph libraries (CuSha, MapGraph, Ligra) and hardwired GPU implementations on a Tesla K40c GPU . All PageRank times are normalized to one iteration. Hardwired GPU implementations for each primitive are Enterprise (BFS) [50], delta-stepping SSSP [18], gpu_BC (BC) [72], and conn (CC) [81]. OOM means out-of-memory. A missing data entry means either there is a runtime error, or the specific primitive for that library is not available.*

We report both runtime and million traversed edges per second (MTEPS) as our performance metrics. Runtime is measured by measuring the GPU kernel running time and MTEPS is measured by recording the number of edges visited during the running, and then divide it by the runtime. When a library does not report MTEPS, we use the following equation to compute it for BFS and BC: $\frac{|E|}{t}$ and $\frac{2\times|E|}{t}$ where E is the number of

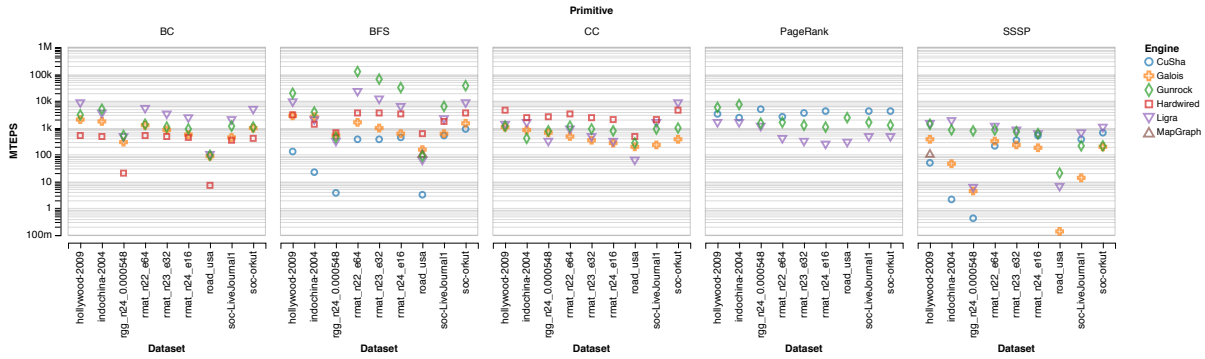


Figure 6.2: *Performance in MTEPS for Gunrock vs. five other graph processing libraries/hardwired algorithms on nine different graph inputs. Data is from Table 6.3.*

edges in the graph and t is runtime. For SSSP, since one edge can be visited multiple times when relaxing its destination node's distance value, there is no accurate way to estimate its MTEPS number. In general, Gunrock's performance on BFS-based primitives (BFS, BC, and SSSP) shows comparatively better results when compared to other graph libraries on seven scale-free graphs (soc-orkut, soc-lj, h09, i04, and rmats), than on two small-degree large-diameter graphs, rgg and roadnet. The primary reason is our load-balancing strategy during traversal (Table 6.5 shows Gunrock's superior performance on warp efficiency, a measure of load-balancing quality, across GPU frameworks and datasets), and particularly our emphasis on good performance for highly irregular graphs. As well, graphs with uniformly low degree expose less parallelism and would tend to show smaller gains in comparison to CPU-based methods. Table 6.4 shows Gunrock's scalability. In general, runtimes scale roughly linearly with graph size for BFS, but primitives with heavy use of atomics on the frontier (e.g., BC, SSSP, and PR) show increased atomic contention within the frontier as graph sizes increase and thus do not scale ideally. For CC, the reason of non-ideal scalability is mainly due to the increase of race conditions when multiple edges try to hook their source node to a common destination node, which happens more often when power-law graph size gets bigger and degree numbers become more unevenly distributed.

vs. CPU Graph Libraries We compare Gunrock's performance with four CPU graph libraries: the Boost Graph Library (BGL) [79], one of the highest-performing CPU single-threaded

graph libraries [54]; PowerGraph, a popular distributed graph library [25]; and Ligra [78] and Galois [61, 68], two of the highest-performing multi-core shared-memory graph libraries. Against both BGL and PowerGraph, Gunrock achieves 6x–337x speedup on average on all primitives. Compared to Ligra, Gunrock's performance is generally comparable on most tested graph primitives; note Ligra uses both CPUs effectively. The performance inconsistency for SSSP vs. Ligra is due to comparing our Dijkstra-based method with Ligra's Bellman-Ford algorithm. Our SSSP's edge throughput is smaller than BFS but similar to BC because of similar computations (atomicMin vs. atomicAdd) and a larger number of iterations for convergence. The performance inconsistency for BC vs. Ligra on four scale-free graphs is because that Ligra applies pull-based traversal on BC while Gunrock has not yet done so. Compared to Galois, Gunrock shows more speedup on traversal-based graph primitives (BFS, SSSP, and BC) and less performance advantage on PageRank and CC due to their dense computation and more regular frontier structures.

vs. Hardwired GPU Implementations and GPU Libraries Compared to hardwired GPU implementations, depending on the dataset, Gunrock's performance is comparable or better on BFS, BC, and SSSP. For CC, Gunrock is 5x slower (geometric mean) than the hardwired GPU implementation due to irregular control flow because each iteration starts with full edge lists in both hooking and pointer-jumping phases. The alternative is extra steps to perform additional data reorganization. This tradeoff is not typical of our other primitives. While still achieving high performance, Gunrock's application code is smaller in size and clearer in logic compared to other GPU graph libraries. Gunrock's Problem class (that defines problem data used for the graph algorithm) and kernel enactor are both template-based C++ code; Gunrock's functor code that specifies per-node or per-edge computation is C-like device code without any CUDA-specific keywords. Writing Gunrock code may require parallel programming concepts (e.g., atomics) but neither details of low-level GPU programming nor optimization knowledge. [1]

[1] We believe this assertion is true given our experience with other GPU libraries when preparing this evaluation section, but freely acknowledge this is nearly impossible to quantify. We invite readers to peruse our annotated code for BFS and SALSA at `http://gunrock.github.io/gunrock/doc/annotated_primitives/annotated_primitives.html`.

Gunrock compares favorably to existing GPU graph libraries. MapGraph is faster than Medusa on all but one test [21] and Gunrock is faster than MapGraph on all tests: the geometric mean of Gunrock's speedups over MapGraph on BFS, SSSP, PageRank, and CC are 4.679, 12.85, 3.076, and 5.69, respectively. Gunrock also outperforms CuSha on BFS and SSSP. For PageRank, Gunrock achieves comparable performance without the G-Shard data preprocessing, which serves as the main load-balancing module in CuSha. The 1-GPU Gunrock implementation has 1.83x more MTEPS (4731 vs. 2590) on direction-optimized BFS on the soc-LiveJournal dataset (a smaller scale-free graph in their test set) than the 2-CPU, 2-GPU configuration of Totem [71]. All three GPU BFS-based high-level-programming-model efforts (Medusa, MapGraph, and Gunrock) adopt load-balancing strategies from Merrill et al.'s BFS [59]. While we would thus expect Gunrock to show similar performance on BFS-based graph primitives as these other frameworks, we attribute our performance advantage to two reasons: (1) our improvements to efficient and load-balanced traversal that are integrated into the Gunrock core, and (2) a more powerful, GPU-specific programming model that allows more efficient high-level graph implementations. (1) is also the reason that Gunrock implementations can compete with hardwired implementations; we believe Gunrock's load-balancing and work distribution strategies are at least as good as if not better than the hardwired primitives we compare against. Gunrock's memory footprint is at the same level as Medusa and better than MapGraph (note the OOM test cases for MapGraph in Table 6.3). The data size is $\alpha|E| + \beta|V|$ for current graph primitives, where $|E|$ is the number of edges, $|V|$ is the number of nodes, and α and β are both integers where α is usually 1 and at most 3 (for BC) and β is between 2 to 8.

Figure 6.3 shows Gunrock (release version v0.4.0)'s performance on four different GPUs: Tesla K40m, Tesla K80, Tesla M40, and Tesla P100. On different GPUs, Gunrock's performance shows great consistency. The performance on Tesla P100 wins because of its superior memory performance and the improvements on atomic operations.

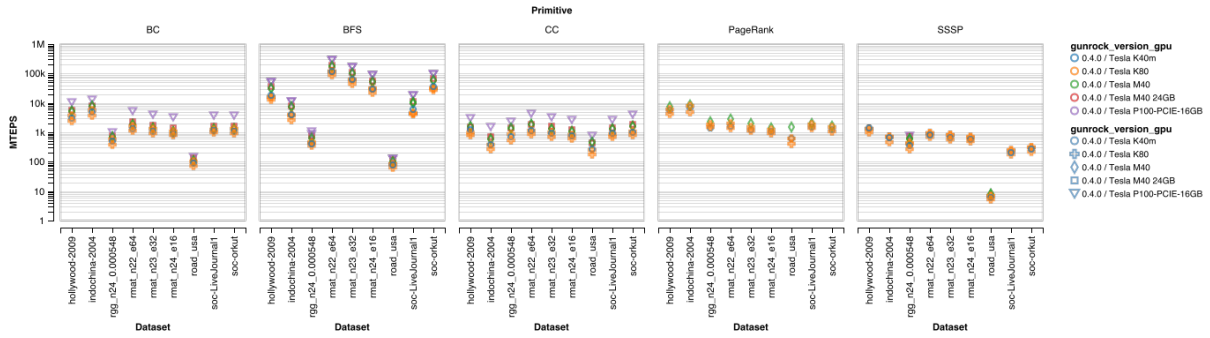


Figure 6.3: *Gunrock's performance on different GPU devices. Tesla M40 24GB is attached to Haswell and has higher boost clock (1328500 Hz), Tesla M40 is attached to Ivy Bridge with a boost clock of 1112000 Hz.*

	Runtime (ms)					Edge throughput (MTEPS)		
Dataset	BFS	BC	SSSP	CC	PageRank	BFS	BC	SSSP
kron_g500-logn18 ($v = 2^{18}, e = 14.5$M)	1.319	12.4	13.02	9.673	3.061	10993	2339	1113
kron_g500-logn19 ($v = 2^{19}, e = 29.7$M)	1.16	26.93	26.59	20.41	6.98	25595	2206	1117
kron_g500-logn20 ($v = 2^{20}, e = 60.6$M)	2.355	67.57	56.22	43.36	19.63	25723	1793	1078
kron_g500-logn21 ($v = 2^{21}, e = 123.2$M)	3.279	164.9	126.3	97.64	60.14	37582	1494	975.2
kron_g500-logn22 ($v = 2^{22}, e = 249.9$M)	5.577	400.3	305.8	234	163.9	44808	1248	817.1
kron_g500-logn23 ($v = 2^{23}, e = 505.6$M)	10.74	900	703.3	539.5	397.7	47077	1124	719

Table 6.4: *Scalability of 5 Gunrock primitives (runtime and edges traversed per second) on a single GPU on five differently-sized synthetically-generated Kronecker graphs with similar scale-free structure.*

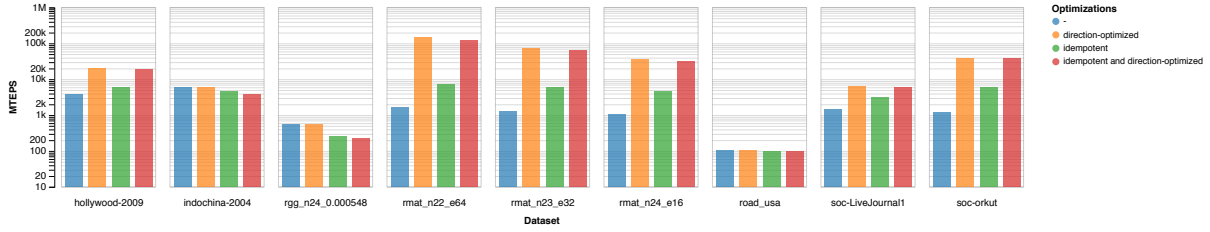


Figure 6.4: *Gunrock's performance with different combinations of idempotence and direction-optimized traversal.*

6.2 Optimization Strategies Performance Analysis

Figure 6.4 and figure 6.5 shows how different optimization strategy combinations and different workload mapping strategies affect the performance of graph traversal; here we use BFS as an

Alg.	Framework	soc-ork	soc-lj	h09	i04	rmat-22	rmat-23	rmat-24	rgg	roadnet
BFS	Gunrock	92.01%	87.89%	86.52%	94.3%	90.11%	91.21%	92.85%	79.23%	80.65%
	MapGraph	—	95.83%	95.99%	—	—	—	—	—	86.59%
	CuSha	69.42%	58.85%	57.2%	52.64%	49.8%	43.65%	42.17%	64.77%	—
SSSP	Gunrock	96.54%	95.96%	94.78%	96.34%	96.79%	96.03%	96.85%	77.55%	80.11%
	MapGraph	—	—	—	—	—	—	—	—	99.82%
	CuSha	—	—	—	—	—	—	—	—	—
PR	Gunrock	99.54%	99.45%	99.37%	99.54%	98.74%	98.68%	98.29%	99.56%	96.21%
	MapGraph	—	98.8%	98.72%	—	—	—	—	—	99.82%
	CuSha	80.13%	64.23%	93.23%	55.62%	79.7%	74.87%	70.29%	75.11%	—

Table 6.5: *Average warp execution efficiency (fraction of threads active during computation). This figure is a good metric for the quality of a framework's load-balancing capability. (— indicates the graph framework failed to run on that dataset.)*

example.

Without losing generality, for our tests on different optimization strategies, we fixed the workload mapping strategy to LB_CULL so that we can focus on the impact of different optimization strategies. Our two key optimizations: idempotence operation and direction-optimized traversal both show performance increases compared to the base-line LB_CULL traversal. Also, our experiment shows that when using LB_CULL for advance, with both direction-optimized and idempotence flags open would always yield worse performance than with only direction-optimized flag open. The reason behind this is that enabling idempotence operation in direction-optimized traversal iteration would cause additional global data access of a visited status bitmask array. We also note that for graphs with very small degree standard deviation (< 5), using idempotence will actually hurt the performance. The reason is for these graphs, the number of common neighbors are so small that avoiding atomic operations will not provide much performance gain, but using idempotence will introduce an additional pass of filter heuristics and the cost of that cancels out the mere performance gain from avoiding atomic operations and even causes a performance decrease.

Our tests on different workload mapping strategies (traversal mode) shows that LB_CULL

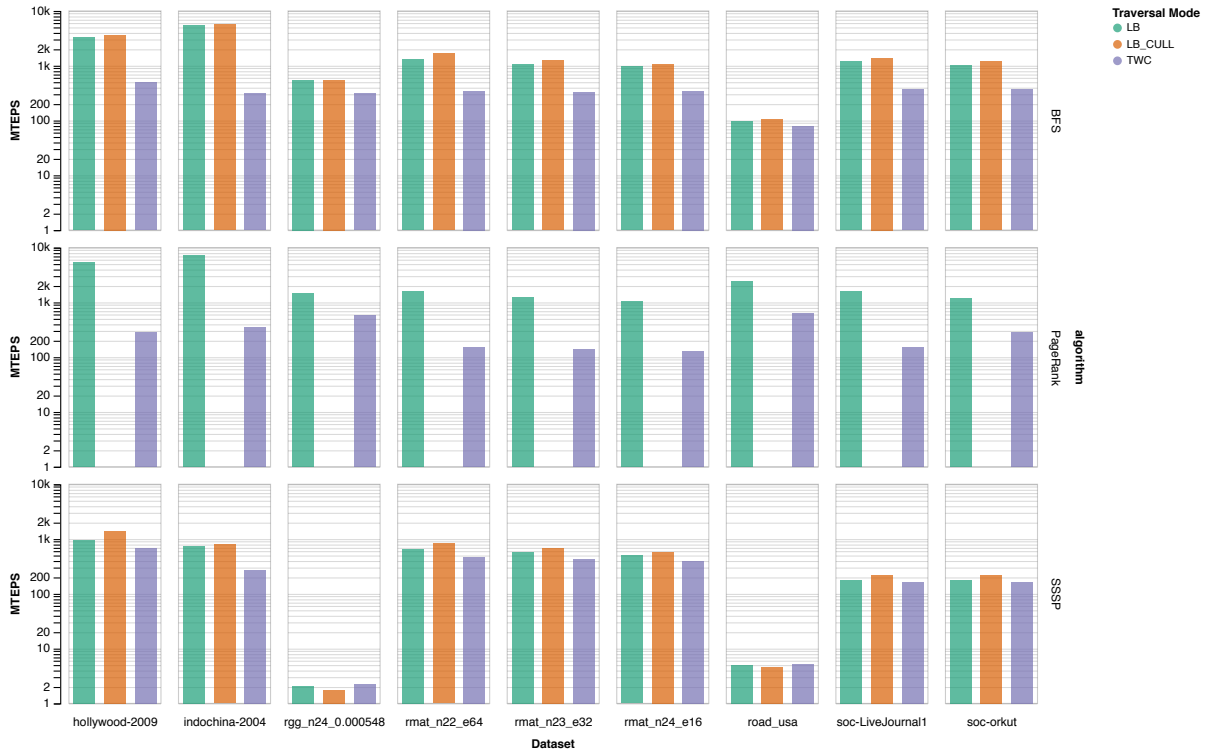


Figure 6.5: *BFS, SSSP, and PR's performance in Gunrock using three different workload mapping streategies: LB, LB_CULL, and TWC.*

constantly outperforms other two strategies on these 9 datasets. Its better performance compared to TWC is due to its switching between load balancing over input frontier and load balancing over output frontier according to the input frontier size, as we have discussed in section 3.2.1.3. Its better performance compared to LB is due to its kernel fusion implementation, which reduces the kernel launch overhead and also some additional data movement between operators. However, without a more thorough performance characterization, we cannot make the conclusion that LB_CULL will always have better performance, as in our SSSP tests on two mesh-like graphs with large diameters and small average degrees, TWC has shown better performance. In general, we currently predict which strategies will be most beneficial based only on the degree distribution; many application scenarios may allow pre-computation of this distribution and thus we can choose the optimal strategies before we begin computation.

Figure 6.6 shows the interesting patterns of performance changing according to direction-optimized parameters' variation. In general, R-Mat graphs , social networks, and web graphs

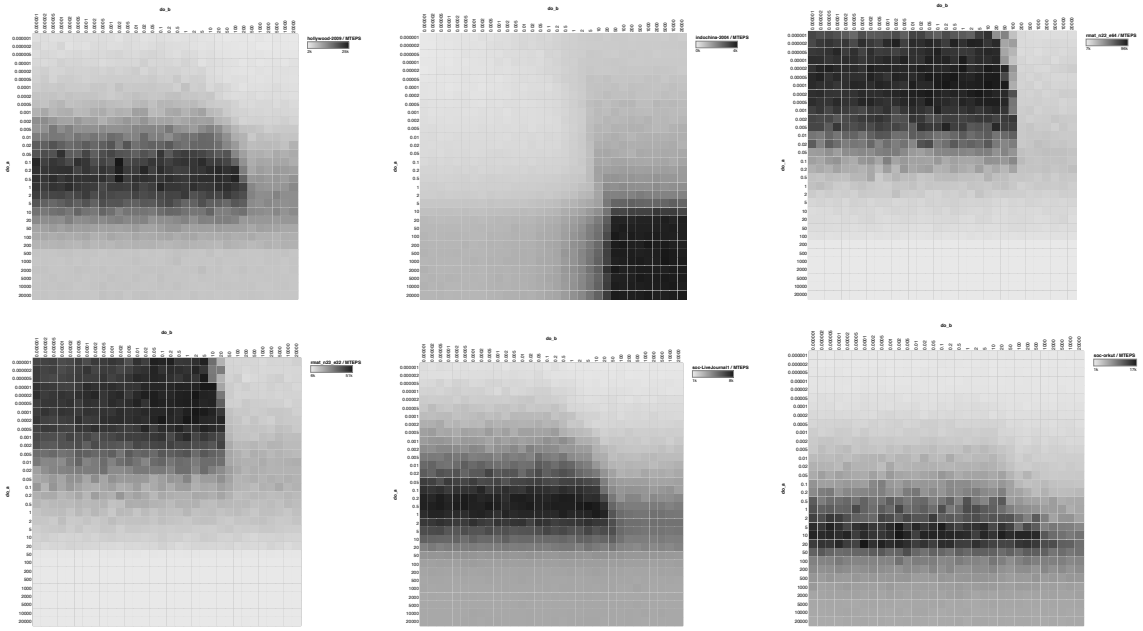


Figure 6.6: *Heatmaps to show how Gunrock's two direction-optimized parameters,* ***do_a****, and* ***do_b*** *affect BFS performance. Each square represents the average of 25 BFS runs, each starting at a random node. Darker color means higher TEPS . Three datasets of top row are: hollywood-2009, indochina-2004, and rmat-n22 (from left to right); three datasets of bottom row are: rmat-n23, soc-livejournal, and soc-orkut (from left to right).*

show different patterns due to their different degree distributions. But every graph has a rectangular region where performance figures show discontinuity from the outside region. This is because our two direction-optimized parameters indirectly affect the number of iterations of push-based traversal and pull-based traversal, which is discrete and thus form the rectangular region. As shown in figure 6.6, with one round of BFS execution that starts with push-based traversal, increasing *do_a* from a small value would speedup the switch from push-based to pull-based traversal, and thus always yield better performance at first. Until when *do_a* is getting to large and causes pull-based traversal to start at a too early iteration, then the performance will drop because usually in early iterations, small sized frontier show better performance on push-based traversal rather than pull-based traversal. Parameter *do_b* controls when to switch back from pull-based to push-based traversal. On most graphs, regardless of whether they are scale-free or mesh-like graphs, keeping a smaller *do_b* so that the switch from pull-based to push-based traversal never happens would help us achieve better performance. The only outlier being indochina-2004, the location of its rectangular region is at the lower right, which means

either keeping both a larger *do_a* and *do_b* will yield better performance on this dataset, or the parameter range we have chosen is not wide enough to show the complete pattern for this dataset.

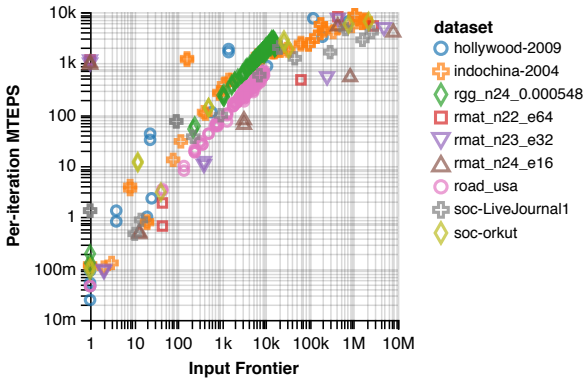


Figure 6.7: *Per-iteration advance performance (in MTEPS) vs. input frontier size.*

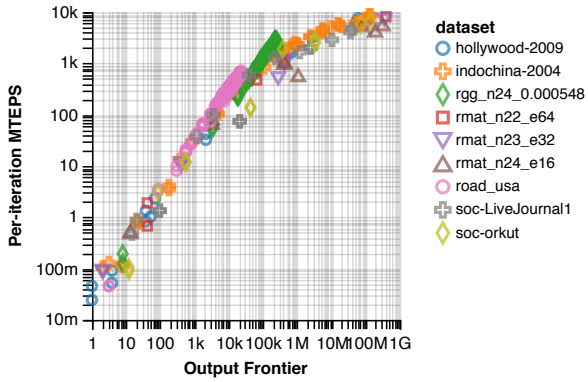


Figure 6.8: *Per-iteration advance performance (in MTEPS) vs. output frontier size.*

Figure 6.7 and figure 6.8 show Gunrock's per-iteration advance performance increase according to the increase of both input and output frontier size. The input frontier figure has more noises because for some scale-free datasets, a small input frontier could generate a very large output frontier, and cause an outlier in the scatter plot. For two roadnet graphs (roadnet and rgg), we are using TWC strategy, while all other seven datasets are using LB_CULL strategy. This creates two different types of curve. For datasets that use LB_CULL strategy, the input frontier performance curve reaches a horizontal asymptote when the input frontier size is larger than 1 million, and the output frontier performance curve keeps growing even for a 1 billion sized output frontier. However, the performance increase for datasets use LB_CULL slows down when the output frontier size is larger than one million. For two datasets that use TWC strategy, both the input frontier performance curve and the output frontier performance curve are linear, and presents better performance than datasets using LB_CULL when the output frontier size is larger than 1 million. This shows that TWC can achieve better performance than LB_CULL on regular graphs with a relatively large frontier. The results of this experiment proves that in general, to achieve good performance, Gunrock needs a relatively large frontier to fully utilize the computation resource of a GPU .

6.3 GPU Who-To-Follow Performance Analysis

The datasets used in our experiments are shown in Table 6.6, and Table 6.7 shows the runtimes of our GPU recommendation system on these datasets. Runtimes are for GPU computation only and do not include CPU-GPU transfer time. The wiki-Vote dataset is a real social graph dataset that contains voting data for Wikipedia administrators; all other datasets are follow graphs from Twitter and Google Plus [45, 49]. Twitter09 contains the complete Twitter follow graph as of 2009; we extract 75% of its user size and 50% of its social relation edge size to form a partial graph that can fit into the GPU's memory.

Table 6.6: *Experimental datasets for GPU Who-To-Follow algorithm.*

Dataset	Vertices	Edges
wiki-Vote	7.1k	103.7k
twitter-SNAP	81.3k	2.4M
gplus-SNAP	107.6k	30.5M
twitter09	30.7M	680M

6.3.1 Scalability

In order to test the scalability of our WTF-on-GPU recommendation system, we ran WTF on six differently-sized subsets of the twitter09 dataset. The results are shown in Figure 6.9. We see that the implementation scales sub-linearly with increasing graph size. As we double the graph

Table 6.7: *GPU WTF runtimes for different graph sizes.*

Time (ms)	wiki-Vote	twitter	gplus	twitter09
PPR	0.45	0.84	4.74	832.69
CoT	0.54	1.28	2.11	51.61
Money	2.70	5.16	18.56	158.37
Total	4.37	8.36	26.57	1044.99

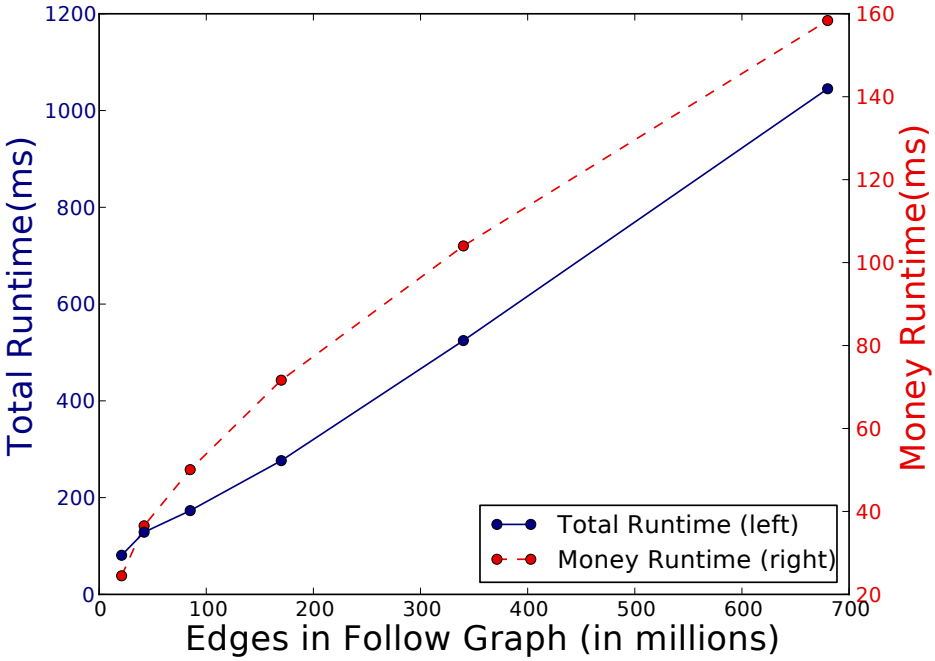


Figure 6.9: *Scalability graph of our GPU recommendation system.*

size, the total runtime increases by an average of 1.684x, and the runtime for Money increases by an average of 1.454x. The reason lies in our work-efficient parallel implementation. By doing per-vertex computation exactly once and visiting each edge exactly once, our parallel algorithm performs linear $O(m+n)$ work. The reason that we have better scalability for the Money algorithm is that although we are doubling the graph size each time, the CoT size is fixed at 1000.

6.3.2 Comparison to Cassovary

We chose to use the Cassovary graph library for our CPU performance comparison. The results of this comparison are shown in Table 6.8. Cassovary is a graph library developed at Twitter. It was the library Twitter used in their first WTF implementation [32]. The parameters and Cassovary function calls for this implementation can be found in the Appendix.

We achieve speedups of up to 1000x over Cassovary for the Google Plus graph, and a speedup of 14x for the 2009 Twitter graph, which is the most representative dataset for the WTF appli-

Table 6.8: *GPU WTF runtimes comparison to Cassovary (C).*

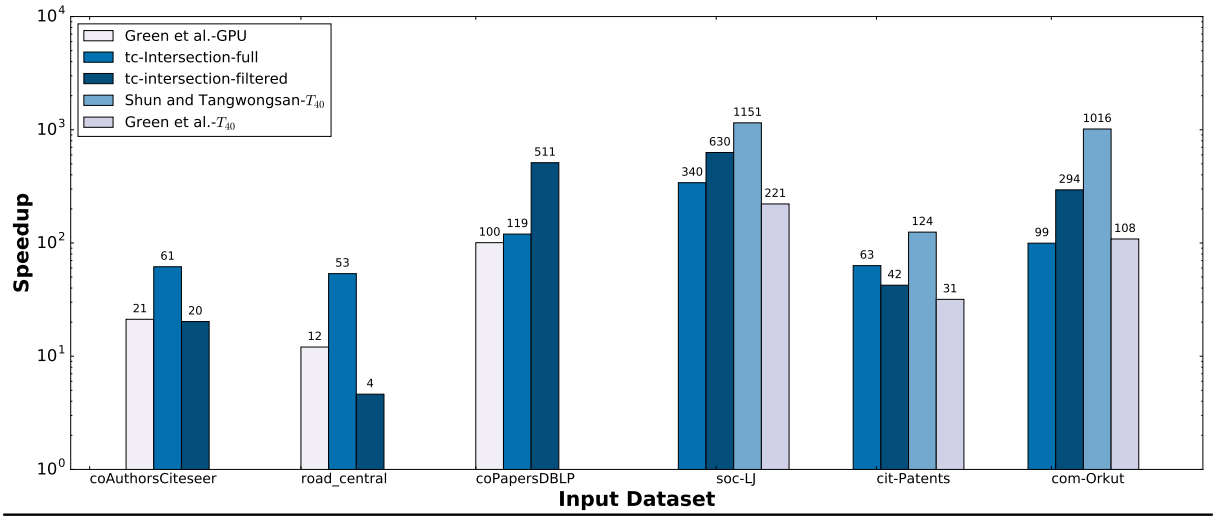
	wiki-Vote		twitter		gplus		twitter09	
Step (runtime)	C	GPU	C	GPU	C	GPU	C	GPU
PPR (ms)	418	0.45	480	0.84	463	4.74	884	832.69
CoT (ms)	262	0.54	2173	1.28	25616	2.11	2192	51.61
Money (ms)	357	2.70	543	5.16	2023	18.56	11216	158.37
Total (ms)	1037	4.37	3196	8.36	28102	26.57	14292	1044.99
Speedup		235.7		380.5		1056.5		13.7

cation. One difference between the GPU algorithm and the Cassovary algorithm is that we used the SALSA function that comes with the Cassovary library, instead of using Twitter's Money algorithm for the final step of the algorithm. Both are ranking algorithms based on link analysis of bipartite graphs, and in the original Who-To-Follow paper [32], Gupta et al. use a form of SALSA for this step, so this is reasonable for a comparison.

6.4 GPU Triangle Counting Performance Analysis

We compare the performance of our GPU TC to three different exact triangle counting methods: Green et al.'s state-of-the-art GPU implementation [29] that runs on an NVIDIA K40c GPU, Green et al.'s multicore CPU implementation [28], and Shun et al.'s multicore CPU implementation [77]. Both of the state-of-the-art CPU implementations are tested on a 40-core shared memory system with two-way hyper-threading; their results are from their publications. Our CPU baseline is an implementation based on the *forward* algorithm by Schank and Wagner [73].

In general, Gunrock's TC implementation shows better performance than the state-of-the-art GPU implementations because of our key optimizations on workload reduction and GPU resource utilization. It achieves comparable performance to the fastest shared-memory CPU TC implementation. Gunrock's TC implementation with only the simple per-thread batch set intersection kernel achieves a 2.51× speedup (geometric-mean) speedup compared to Green et



Dataset	coAuthorsCiteseer	road_central	coPapersDBLP	soc-LJ	cit-Patents	com-Orkut
time (s)	0.28	3.254	100.7	564.3	9.856	1951

Figure 6.10: *Execution-time speedup (top figure) for our Gunrock TC implementations ("tc-intersection-full" and "tc-intersection-filtered"), Green et al.'s GPU implementation [29] ("Green et al.-GPU"), Shun and Tangwongsan's 40-core CPU implementation [77] ("Shun and Tangwongsan-T_{40}") and Green et al.'s 40-core CPU implementation [28] ("Green et al.-T_{40}"). All are normalized to a baseline CPU implementation [73] on six different datasets. Baseline runtime (in seconds) given in the table.*

al.'s CPU implementation [28] and a 4.03× geomean speedup compared to Green et al.'s GPU implementation [29]. We believe our speedup is the result of two aspects of our implementation: using filtering in edge list generation and reforming the induced subgraph with only the edges not filtered which reduces five-sixths of the workload, and dynamic grouping helps maintain a high GPU resource utilization. In practice we observe that for scale-free graphs, the last step of optimization to reform the induced subgraph show a constant speedup over our intersection method without this step. However, for road networks and some small scale-free graphs, the overhead of using segmented reduction will cause a performance drop. We expect further performance gains from future tuning of launch settings for the large-neighbor-list intersection kernel, which we do not believe is quite optimal yet. These optimizations are all transparent to our data-centric programming model and apply to segmented intersection graph operator which could potentially speedup other graph primitives.

Chapter 7

Conclusion

Our work on the data-centric programming model for graph processing on the GPU has brought us several steps closer to a highly programmable high-performance graph analytics system, it also opens up various interesting yet challenging research opportunities. This section will conclude our dissertation with current limitations of our work, future works, and a short summary.

7.1 Limitations

Single-GPU: Current data-centric programming model is designed and optimized for single-GPU architecture. There is a single-node, multi-GPU extension [65], but we haven't looked beyond that to multiple nodes [67].

Fits-in-memory: Current single-GPU architecture does not allow graph analytics if the graph size does not fit in a single-GPU's global memory.

Static Graph: Current data-centric programming model is designed to process static graphs. We haven't worked on supporting streaming graphs and graphs with dynamically changing topology.

Limited Compiler Optimizations: Current library design does not allow extensive compiler optimizations such as kernel fusion and AST (abstract syntax tree) generation [64].

7.2 Future Work

Moving forward, there are several aspects we could improve over the current Gunrock, including architecture, execution model, meta-linguistic abstraction, performance characterization, core graph operators, new graph primitives, graph datatype, and usability. We will expand each item with details in this section.

7.2.1 Architecture and Execution Model

7.2.1.1 Architecture

To enable large-scale graph analysis where the data do not fit in GPU device memory, we need to scale better to address larger datasets with multiple techniques: multi-GPU on a single node, on multiple nodes, on out-of-core data, and on streaming data.

Scale-Out: Gunrock is designed to serve as a general framework which can be the core component of a multi-GPU single-node graph processing system [65]. Designing one such system raises interesting research problems in two directions: 1) the impact of different partitioning methods on the performance; 2) computation-communication trade-offs for the inter-GPU data exchange. We would like to eventually go towards multi-GPU and multi-node graph processing system. Apart from the two research questions for Multi-GPU on a single node we need to answer, there are several research directions we would like to explore such as dynamic repartitioning for load balancing, heterogeneous computing for huge graphs with large diameter (long-tail effect), and vertex cut.

Scale-Up: CPU-GPU memory bandwidth is still limited compared to intra-GPU and GPU-GPU bandwidth, thus we need to think under what situation out-of-core is necessary and/or useful. The current Gunrock framework has not been designed to support out-of-core, there are several works [46, 76] we would like to study to make improvements in Gunrock in terms of graph data representation, partitioning, and communication reduction to make it fit this architecture.

Streaming: Streaming has been an important computing model for massively data processing. There are some recent works on mapping the streaming model to GPUs by partitioning graph into edge streams and processing the graph analytics in parallel [69, 75]. It requires a more general architectural support than out-of-core and raises interesting research questions such as how to partition the graph and when to synchronize. At its core, Gunrock's data-centric programming model maps naturally to streaming process where a frontier is an edge stream. However, like in the case of Scale-up, we still need further study other frameworks to make our own design choices.

7.2.1.2 Execution Model

An asynchronous execution model and higher-level task parallelism are two potentially interesting directions.

Asynchronous: The work of Wang et al. [86] views an asynchronous execution model in BSP as the relaxation of two synchrony properties: 1) isolation, meaning within an iteration, newly generated messages from other vertices can not be seen; and 2) consistency, meaning each thread waits before it proceeds to the next iteration until all other threads have finished the process for the same iteration. To enable asynchronous execution on current Gunrock, we could 1) use a priority queue in single-node GPU, allow merging multiple remote frontiers in multi-node GPU, or adding multi-pass micro iterations within one iteration to relax the consistency property; and 2) allow per-vertex/per-edge operations to propagate results to others to relax the isolation property. In terms of the impact to our data-centric programming model, asynchronous execution could significantly accelerate convergence in primitives such as SSSP, CC, PR, and LP, by intelligently ordering vertex updates and incorporating the most recent updates.

Higher-level task parallelism: Higher-level task parallelism can bring more parallelism for several graph primitives. Preliminary work [51, 55] in this direction executes multiple BFS passes simultaneously on GPUs. Applications benefit from adding this additional level of parallelism on streaming multiprocessor level include all-pairs shortest paths and

BC . Usually such higher-level task parallelism requires keeping multiple active frontiers, and specify different group of blocks to handle graph workload on different active frontiers. Gunrock's data-centric abstraction can be used to implement this with some additional modifications on frontier management and workload mapping.

7.2.2 Meta-Linguistic Abstraction

Gunrock as a Backend: With the current Gunrock implementation, we have a set of C/C++-friendly interfaces which makes Gunrock callable from Python, Julia, and many other high-level programming languages. The future work along this path is to provide Gunrock's computation power to other higher level graph analytics framework (such as TinkerPop or NetworkX) or create our own graph query/analysis DSL on top of Gunrock.

Add More Backends to Gunrock: Gunrock is not the only way to implement our data-centric programming model, an interesting research direction is to explore the possibility to make other back ends support data-centric programming model as well. Such back ends include GraphBLAS (a matrix-based graph processing framework) [42], VertexAPI2 (a GAS-style graph processing framework) [20], and the EmptyHeaded Engine [1] (a Boolean algebra set-operation-based graph query library).

7.2.3 Core Graph Operators

Every graph analytics framework has its own abstraction represented as a set of supported operators on the graph. The design choice is related to hardware architecture and programming language. An interesting research question for graph analytics on the GPU is: What is the right set of graph operators that Gunrock should support? From the original advance, filter, and compute operator set, we have already expanded to include neighborhood reduction and segmented intersection, However, there are still more graph operators which are potentially desirable for adding higher-level, more complex graph primitives into Gunrock:

Priority Queue Using multi-split [2], we can create priority queues with more than one priority

level. This can be applied to the delta-stepping SSSP algorithm and can also serve as an alternative frontier manipulation method in an asynchronous execution model.

Sampling A sampling operator can be viewed as an extension to the standard filter. Currently its applications include approximated BC, TC, and more approximated graph primitives.

Forming Supervertex In our current MST primitive, we have implemented a supervertex forming phase using a series of filter, advance, sort, and prefix-sum. This could be integrated into a new operator. Note that currently, we form supervertices by rebuilding a new graph that contains supervertices. This can be used in hierarchical graph algorithms such as clustering and community detection.

An alternative direction would be to offer a set of lower-level data-parallel primitives including both general primitives such as prefix-sum and reduction, as well as graph-specific primitives such as load-balanced search (for evenly distributing edge expanding workload), and use them to assemble our higher-level graph operators. In Chapter 4, we did experiments of building up graph operators using moderngpu's transform primitives in Mini Gunrock.

Which of the two approaches—implementing many graph operators and specific optimization for each operator, and building up graph operators on top of lower-level data-parallel primitives—do we use in a GPU graph analytics programming model, will affect both the programmability and performance of the system. In Gunrock, we are using a mixed model where we share common components such as load-balanced search, prefix-sum, and compact for advance, filter and neighborhood reduction, while we also include specialized optimizations for advance, filter, and segmented intersection. An interesting future research direction is to think of the graph operator abstraction at a more meta level, and try to build up a hierarchical framework for designing graph operators which includes not only graph traversal and computation building blocks such as all five graph operators we have in Gunrock today, but also memory and workload optimization building blocks which can be plugged into and replaced in our higher-level graph operator implementation.

7.2.4 New Graph Primitives

Our current set of graph primitives are mostly textbook-style algorithms. One future work is to expand this list to more complex graph primitives spanning from network analysis, machine learning, data mining, to graph queries.

Graph Coloring and Maximal Independent Set Efficient graph matching and coloring algorithms [17] can be computed by Gunrock more efficiently than previous work. Maximal independent set (MIS) algorithms follow the same path. These primitives can serve as the building blocks for asynchrony and task-level parallelism.

MIS can be implemented within the Gunrock framework by having every node use a hash function to generate a random number. An advance is performed from each node to determine whether its random number is a local maximum or not. If it is, then it is marked as belonging to the independent set. A second advance is launched these newly marked nodes to mark the neighbors of the independent set as not belonging to the independent set.

As it stands, the Gunrock framework can implement this algorithm. However, a new operator called neighborhood reduction that is a kernel fusion of advance and compute operators can be implemented to perform this operation more efficiently.

In the case of graph coloring, every iteration only requires one advance that colors the local maxima a certain color. However, graph coloring would benefit from the neighborhood reduce operation as well.

One optimization that can be performed is when the unvisited vertices frontier is large. Similar for SSSP (see section 5.1.2), it is possible to use Gunrock's two-level priority queue strategy to improve work-efficiency at the cost of parallelism. This can be done by considering nodes whose random numbers are above a certain threshold as high priority. Nodes whose random numbers that fall below the threshold are counted as low priority.

Strongly Connected Component A directed graph is strongly connected if any node can reach

any other node. This algorithm is traditionally a problem related to depth-first search, which is considered unsuitable for GPUs. However, Slota et al.'s work [80] improved on Barnat et al.'s DFS-based work [3] and avoids DFS by combining BFS and graph coloring.

More Traversal-based Algorithms To make use of our efficient graph traversal operators, we can modify our BFS to implement several other similar primitives: 1) st-connectivity, which simultaneously processes two BFS paths from s and t; 2) A* search as a BFS-based path finding and search algorithm; 3) Edmonds-Karp as a BFS-based maximum flow algorithm; 4) {reverse} Cuthill-McKee algorithm, a sparse matrix bandwidth reduction algorithm based on BFS; 5) belief propagation, a sum-product message passing algorithm based on BFS; and finally 6) radii estimation algorithm, a k-sample BFS-based algorithm for estimating the maximum radius of the graph.

Subgraph Isomorphism Recent subgraph matching in large graph databases works on the GPU [83] and CPUs [11, 33, 82] both of which use graph traversal extensively, and thus could fit nicely in Gunrock's framework. But when mapping them onto GPUs, these methods are all memory-bounded which drives us to build more effective filtering and joining methods. One optimization that can be used in the joining phase is k-way intersection for joining multiple candidate edges at the same time. So we can extend the existing 2-way intersection operator for triangle counting in Gunrock. Other than that, we want to extend the implementation to support programmable input in terms of what people are searching for. In that case, we need to build a parser to transform that format into the graph format that we currently support.

k-SAT k-SAT is a type of Boolean satisfiability problem which can be represented as an iterative updating process on a bipartite graph. Combining the sampling operator and our efficient traversal operator, it potentially fits nicely into Gunrock.

SGD and MCMC Both Stochastic Gradient Descent in matrix completion and Markov Chain Monte Carlo have large amounts of parallelism. The former can be presented as either an iterative bipartite graph traversal-updating algorithm [40] or a conflict-free subgraph

parallel updating algorithm. In order to decrease the number of conflict updates and increase the data re-usage rate, the incomplete matrix is divided into sub-matrices and graph coloring will help identify the conflict-free elements in those small sub-matrices. Graph coloring can take advantage of Gunrock's efficient traversal operator to achieve better performance. MCMC's iterative dense update makes it a good candidate for matrix-based graph algorithms. However, Gunrock operators can also represent this problem.

7.2.5 New Graph Datatypes

All graph processing systems on the GPU are using CSR and edge list internally, moving forward, there needs to be innovations in graph datatypes to enable graph primitives on mutable graphs and matrix-typed graph primitives.

Mutable Graphs: The meaning of mutable is twofold: mutable by primitive, and mutable by input data. *Mutable by algorithm* means that the graph primitive changes the graph structure, and includes primitives such as MST, community detection, mesh refinement, and Karger's mincut. The operations related include simple ones such as adding/removing nodes/edges, and more complex ones such as forming supervertex. Currently there is no good solution for handling general graph mutations on GPUs with efficiency. *Mutable by input data* means that we process our algorithms on input datasets that change over time (so we'd like to both incrementally update the graph data structure and incrementally update a solution given the incremental change in the data structure). We need to provide either approximated results or the capability of doing incremental computation.

Adjacency Matrix Form: Matrix-based graph algorithms are also widely used for graph processing such as BFS and PageRank. Our sparse-matrix sparse-vector operator and its application in BFS [91] has shed light on more applications in the form of matrix such as MIS, PageRank, SSSP, and several spectral methods, which are used in algorithms like collaborative filtering.

Optimized Graph Topology Format: The work of Wu et al. [89] shows how data reorganiza-

tion can help with memory uncoalescing. However, for graph analytics, we need to design better strategies since the memory access indices (node IDs in the frontier) are changing dynamically every iteration. Both CuSha [43] and Graphicionado [16] proposed optimizations on memory access via edge-list grouping according to source node ID and destination node ID. Such optimization and a well-designed cache framework would reduce random memory access during graph analytics. It is also an open question if bandwidth reduction algorithm such as reverse Cuthill-McKee algorithm will bring better memory locality for graph analytics.

Rich-data-on-{vertices,edges}: Complex networks often contain rich data on vertices and/or edges. Currently Gunrock puts all the information on edge/vertex into a data structure on the GPU , which is not ideal. Adding the capability of loading partial edge/vertex information onto the GPU could enable graph query tasks and several network analysis primitives which use these rich information during their algorithms.

7.3 Summary

Gunrock was born when we spent two months writing a single hardwired GPU graph primitive. We knew that for GPUs to make an impact in graph analytics, we had to raise the level of abstraction in building graph primitives. With the work of this dissertation, we show that with appropriate high-level programming model and low-level optimizations, parallel graph analytics on the GPU can be both simple and efficient. More specifically, this dissertation has achieved its two high-level goals:

- Our data-centric, frontier-focused programming model has proven to map naturally to the GPU, giving us both good performance and good flexibility. We have also found that implementing this abstraction has allowed us to integrate numerous optimization strategies, including multiple load-balancing strategies for traversal, direction-optimal traversal, and a two-level priority queue. The result is a framework that is general (able to implement numerous simple and complex graph primitives), straightforward to program (new primitives

only take a few hundred lines of code and require minimal GPU programming knowledge), and fast (on par with hardwired primitives and faster than any other programmable GPU graph library).

- Our open-sourced GPU graph processing library Gunrock provides a graph analytics framework for three types of users: 1) data scientists who want to take the advantage of the GPU's superior computing power in big data applications; 2) algorithm designers who want to use the existing efficient graph operators in Gunrock to create new graph algorithms and applications; and 3) researchers who want to reproduce the results of our research, or make improvements to our core components. We hope that in the future, Gunrock could serve as a standard benchmark for graph processing on the GPU.

References

[1] Christopher R. Aberger, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: Boolean algebra based graph processing. *CoRR*, abs/1503.02368, 2015. URL http://arxiv.org/abs/1503.02368.

[2] Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. GPU multisplit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2016, pages 12:1–12:13, March 2016. doi: 10.1145/2851141.2851169. URL http://www.escholarship.org/uc/item/346486j8.

[3] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on cuda. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 544–555, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.59.

[4] Sean Baxter. Modern gpu introduction. https://nvlabs.github.io/moderngpu/intro.html, 2013.

[5] Sean Baxter. Modern gpu multisets. https://nvlabs.github.io/moderngpu/sets.html, 2013.

[6] Sean Baxter. Moderngpu: Patterns and behaviors for GPU computing. http://moderngpu.github.io/moderngpu, 2013–2016.

[7] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, November 2012. doi: 10.1109/SC.2012.50.

[8] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. doi: 10.1145/1572769.1572795.

[9] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge,

MA, USA, 1990. ISBN 0-262-02313-X.

[10] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi: 10.1080/0022250X.2001.9990249.

[11] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *Proceedings of the 2010 International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '10, pages 248–255, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4138-9. doi: 10.1109/ASONAM.2010.80.

[12] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization*, IISWC-2012, pages 141–151, November 2012. doi: 10.1109/IISWC.2012.6402918.

[13] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12, pages 141–151, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-4531-6. doi: 10.1109/IISWC.2012.6402918.

[14] F. Busato and N. Bombieri. BFS-4K: An efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, July 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2330597.

[15] Daniel Cederman and Philippas Tsigas. On dynamic load-balancing on graphics processors. In *Graphics Hardware 2008*, pages 57–64, June 2008. doi: 10.2312/EGGH/EGGH08/057-064.

[16] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, Washington, DC, USA, 2016. IEEE Computer Society.

[17] Jonathan Cohen and Patrice CastonGuay. Efficient graph matching and coloring on the GPU. *GPU Technology Conference*, March 2012. URL

http://on-demand.gputechconf.com/gtc/2012/presentations/S0332-Efficient-Graph-Matching-and-Coloring-on-GPUs.pdf.

[18] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2014, pages 349–359, May 2014. doi: 10.1109/IPDPS.2014.45. URL http://escholarship.org/uc/item/8qr166v2.

[19] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73:940–952, September 2010. doi: 10.1016/j.jpdc.2012.02.007.

[20] Erich Elsen and Vishal Vaidyanathan. A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction, 2013. http://www.github.com/RoyalCaliber/vertexAPI2.

[21] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of the Workshop on GRAph Data Management Experiences and Systems*, GRADES '14, pages 2:1–2:6, June 2014. doi: 10.1145/2621934.2621936.

[22] Afton Geil, Yangzihao Wang, and John D. Owens. WTF, GPU! Computing Twitter's who-to-follow on the GPU. In *Proceedings of the Second ACM Conference on Online Social Networks*, COSN '14, pages 63–68, October 2014. doi: 10.1145/2660460.2660481. URL http://escholarship.org/uc/item/5xq3q8k0.

[23] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. Efficient large-scale graph processing on hybrid CPU and GPU systems. *CoRR*, abs/1312.3018(1312.3018v2), December 2014.

[24] Ashish Goel, Pankaj Gupta, John Sirois, Dong Wang, Aneesh Sharma, and Siva Gurumurthy. The Who-To-Follow system at Twitter: Strategy, algorithms, and revenue impact. *Interfaces*, 45(1):98–107, February 2015. ISSN 0092-2102. doi: 10.1287/inte.2014.0784.

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Pow-

erGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 17–30. USENIX Association, October 2012.

[26] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685096.

[27] Oded Green, Robert McColl, and David A. Bader. GPU merge path: A GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304621.

[28] Oded Green, Lluís-Miquel Munguía, and David A. Bader. Load balanced clustering coefficients. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 3–10, 2014. ISBN 978-1-4503-2654-4. doi: 10.1145/2567634.2567635.

[29] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the GPU. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '14, pages 1–8, 2014. ISBN 978-1-4799-7056-8. doi: 10.1109/IA3.2014.7.

[30] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[31] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 16–25, June 1994. doi: 10.1145/181014.181021.

[32] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. WTF: The who to follow service at Twitter. In *Proceedings of the International Conference*

on the World Wide Web, pages 505–514, May 2013.

[33] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465300.

[34] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, December 2007. Springer-Verlag. doi: 10.1007/978-3-540-77220-0_21.

[35] Mark Harris. Optimizing parallel reduction in cuda, 2007. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[36] Mark Harris, John D. Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. CUDPP: CUDA data parallel primitives library. http://cudpp.github.io/, 2009–2016.

[37] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, February 2011. doi: 10.1145/1941553.1941590.

[38] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle 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 XVII, pages 349–362, March 2012. doi: 10.1145/2189750.2151013.

[39] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart. Edge v. node parallelism for graph centrality metrics. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 2, pages 15–28. Morgan Kaufmann, October 2011. doi: 10.1016/B978-0-12-385963-1.00002-2.

[40] Rashid Kaleem, Sreepathi Pai, and Keshav Pingali. Stochastic gradient descent on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU

2015, pages 81–89, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3407-5. doi: 10.1145/2716282.2716289.

[41] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, September 2011. doi: 10.1109/MM.2011.89.

[42] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Jose Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *Proceedings of the IEEE High Performance Extreme Computing Conference*, September 2016. URL http://arxiv.org/abs/1606.05790.

[43] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, June 2014. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600227.

[44] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324140.

[45] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the International Conference on the World Wide Web*, pages 591–600, April 2010. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751.

[46] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387884.

[47] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4): 831–838, October 1980. ISSN 0004-5411. doi: 10.1145/322217.322232.

[48] R. Lempel and S. Moran. SALSA: The stochastic approach for link-structure analysis.

ACM Transactions on Information Systems, 19(2):131–160, April 2001.

[49] Jure Leskovec. SNAP: Stanford large network dataset collection. `http://snap.stanford.edu/data/`, 2009–2016.

[50] Hang Liu and H. Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 68:1–68:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: `10.1145/2807591.2807594`.

[51] Hang Liu, H. Howie Huang, and Yang Hu. iBFS: Concurrent breadth-first search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 403–416, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: `10.1145/2882903.2882959`.

[52] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Twenty-Sixth Annual Conference on Uncertainty in Artificial Intelligence*, UAI-10, pages 340–349, July 2010.

[53] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, June 2010. doi: `10.1145/1807167.1807184`.

[54] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, February 2014. doi: `10.1145/2567634.2567638`.

[55] A. McLaughlin and D. A. Bader. Fast execution of simultaneous breadth-first searches on sparse graphs. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 9–18, Dec 2015. doi: `10.1109/ICPADS.2015.10`.

[56] A. McLaughlin, J. Riedy, and D. A. Bader. A fast, energy-efficient abstraction for simultaneous breadth-first searches. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2015. doi: `10.1109/HPEC.2015.7322466`.

[57] Adam McLaughlin and David A. Bader. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC14, pages 572–583, November 2014. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.52.

[58] Duane Merrill. Cub: Flexible library of cooperative threadblock primitives and other utilities for CUDA kernel programming. https://nvlabs.github.io/cub/, 2013–2016.

[59] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, February 2012. doi: 10.1145/2145816.2145832.

[60] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, October 2003. doi: 10.1016/S0196-6774(03)00076-2. 1998 European Symposium on Algorithms.

[61] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, November 2013. doi: 10.1145/2517349.2522739.

[62] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2016. Version 8.0.

[63] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

[64] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. *SIGPLAN Not.*, 51(10):1–19, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984015.

[65] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-GPU graph analytics. *CoRR*, abs/1504.04804(1504.04804v3), April 2016.

[66] Pushkar R. Pande and David A. Bader. Computing betweenness centrality for small world networks on a GPU. In *2011 IEEE Conference on High Performance Embedded Computing*,

Sept 2011.

[67] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.50.

[68] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, June 2011. doi: 10.1145/1993498.1993501.

[69] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[70] Semih Salihoglu and Jennifer Widom. HelP: High-level primitives for large-scale graph processing. In *Proceedings of the Workshop on GRAph Data Management Experiences and Systems*, GRADES '14, pages 3:1–3:6, June 2014. doi: 10.1145/2621934.2621938.

[71] Scott Sallinen, Abdullah Gharaibeh, and Matei Ripeanu. Accelerating direction-optimized breadth first search on hybrid architectures. *CoRR*, abs/1503.04359(1503.04359v1), March 2015.

[72] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, March 2013. doi: 10.1145/2458523.2458531.

[73] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA'05, pages 606–609, 2005. ISBN 3-540-25920-1, 978-3-540-25920-6. doi: 10.1007/11427186_54.

[74] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primi-

tives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, August 2007. doi: 10.2312/EGGH/EGGH07/097-106. URL http://www.idav.ucdavis.edu/publications/print_pub?pub_id=915.

[75] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. GStream: A graph streaming processing method for large-scale graphs on GPUs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 253–254, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688526.

[76] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. Optimization of asynchronous graph processing on GPU with hybrid coloring model. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 271–272, February 2015. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688542.

[77] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE 31st International Conference on Data Engineering*, pages 149–160, April 2015. doi: 10.1109/ICDE.2015.7113280.

[78] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, February 2013. doi: 10.1145/2442516.2442530.

[79] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, December 2001.

[80] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, May 2014. doi: 10.1109/IPDPS.2014.64.

[81] Jyothish Soman, Kothapalli Kishore, and P J Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed*

Processing, Workshops and PhD Forum, IPDPSW 2010, pages 1–8, April 2010. doi: 10.1109/IPDPSW.2010.5470817.

[82] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012. ISSN 2150-8097. doi: 10.14778/2311906.2311907.

[83] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. Fast subgraph matching on large graphs using graphics processors. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications*, volume 9049 of *Lecture Notes in Computer Science*, pages 299–315. Springer International Publishing, 2015. ISBN 978-3-319-18119-6. doi: 10.1007/978-3-319-18120-2_18.

[84] Stanley Tzeng, Brandon Lloyd, and John D. Owens. A GPU task-parallel model with dependency resolution. *IEEE Computer*, 45(8):34–41, August 2012. doi: 10.1109/MC.2012.255. URL http://www.idav.ucdavis.edu/publications/print_pub?pub_id=1091.

[85] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181.

[86] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*. www.cidrdb.org, 2013. URL http://dblp.uni-trier.de/db/conf/cidr/cidr2013.html#WangXDG13.

[87] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the 1st High Performance Graph Processing Workshop*, HPGP '16, pages 1–8, May 2016. doi: 10.1145/2915516.2915521. URL http://escholarship.org/uc/item/9hf0m6w3.

[88] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2016, March 2016. doi: 10.1145/2851141.2851145. URL

http://escholarship.org/uc/item/6xz7z9k0.

[89] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442523.

[90] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens. Performance characterization for high-level programming models for GPU graph analytics. In *IEEE International Symposium on Workload Characterization*, IISWC-2015, pages 66–75, October 2015. doi: 10.1109/IISWC.2015.13. URL http://escholarship.org/uc/item/2t69m5ht.

[91] Carl Yang, Yangzihao Wang, and John D. Owens. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Graph Algorithms Building Blocks*, GABB 2015, pages 841–847, May 2015. doi: 10.1109/IPDPSW.2015.77. URL http://escholarship.org/uc/item/1rq9t3j3.

[92] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014. doi: 10.1109/TPDS.2013.111.