Performance Characterization of High-Level Programming Models for GPU Graph Analytics

By

YUDUO WU
B.S. (Macau University of Science and Technology, Macau) 2013

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair Dr. John D. Owens, Chair

Dr. Venkatesh Akella

Dr. Nina Amenta

Committee in Charge

2015

To my family . . .

Contents

List of Figures

List of Tables

ABSTRACT

Performance Characterization of High-Level Programming Models for GPU Graph Analytics

In this thesis, we identify several factors that are critical to high-performance GPU graph analytics: efficient and flexible building block operators, synchronization and data movement, workload distribution and load balancing, and memory access patterns. We analyze the impact of these critical factors through three high-level GPU graph analytic frameworks, VertexAPI2, MapGraph, and Gunrock. We also examine their effect on different workloads: four common graph primitives from multiple graph application domains, evaluated through real-world and synthetic graphs. We show that efficient building block operators enable more powerful operations for fast information propagation and result in fewer device kernel invocations, less data movement, and fewer global synchronizations, and thus are key focus areas for efficient large-scale graph analytics on GPUs.

ACKNOWLEDGMENTS

First and foremost, I thank my major advisor Professor John D. Owens for the vital guidance and inspiration that he gave me during my graduate studies. He introduced me to parallel GPU computing, taught me tons of knowledge necessary for researching on large-scale GPU graph analytics and writing this thesis, and gave me many excellent opportunities to meet and interact with industry and outstanding researchers. I am deeply grateful to John for spending hours every week with me editing papers, answering questions and investigating our project.

I would like to thank all my thesis committee members: Professor Venkatesh Akella and Professor Nina Amenta for their valuable comments, detailed editings and encouragement, which gave me an opportunity to improve my research and writing. During my graduate studies, I also had opportunities to interact with many other faculty members at UC Davis, such as Professor Yong Jae Lee, Professor S. Felix Wu, and many others, from them, I learned how to become a more effective graduate student and a student researcher.

In addition, I am also grateful to the many graduate students of Professor Owens who I am fortunate to consider my friends. Calina Copos, Afton Geil, Jason Mak, Kerry Seitz, Vehbi Eşref Bayraktar, Saman Ashkiani, Collin McCarthy, and Andy Riffel. Especially, I would like to thank all Gunrock team members: Yangzihao Wang, Yuechao Pan, Carl Yang, and Leyuan Wang, for their constant support and awesome work. I consider myself to be extremely lucky to have had their support during my graduate studies, and I look forward to continuing our friendship in the years to come.

I wish to express my gratitude to my wonderful girlfriend, Zejun Huang. Her constant accompany and encouragement enabled me to finish writing this thesis. I am very grateful to my parents, Congzhong Wu and Qige Liu, and other family members, for always motivating me to pursue better life and being there whenever I needed them. Without them, I would not have come this far.

Chapter 1

Introduction

1.1 Background and Motivation

Large-scale graph structures with millions or billions of vertices and edges are rapidly created by modern applications to map relationships among high volumes of connected data. Graph analytics, such as traversal, query, matching, and others, are essential components of big-data analytics for many high-performance computing and commercial applications across domains from social network analysis, financial services, scientific simulations, recommendation system, and biological networks to targeted advertising. Nowadays, specialized graph analytic systems including graph search [3], betweenness centrality [4], shortest path findings [5], personalized PageRank [6], and strongly connected components [7] gained significant attention in high-performance computing community as well.

The demands of diverse graph applications and strong desire for efficiency have led to numerous efforts on speeding up graph primitives via parallelization architectures. A number of shared memory CPU graph processing engines have emerged in recent several years, such as GraphChi [8], Galois [9], X-Stream [10], TurboGraph [11], Ligra(+) [12, 13], and Polymer [14]. Prior research focuses on scaling graph-analytics on distributed environments including the Parallel Boost Graph Library (PBGL) [15, 16], Google's Pregel [17], GraphLab [18, 19], PowerGraph [20], GraphX [21], and more. Other work HelP [22], provides a higher level abstraction on top of the GraphX, improves programmability by capturing commonly appearing operations in graph primitives, with the motivation of delivering a set of more intuitive operators to help implement graph primitives faster than low-level implementation or using other existing frameworks. These graph processing frameworks allow programmers to concentrate on expressing primitives, because the framework takes care of automatically scaling the computations on parallel architectures.

1.2 GPU Graph Analytics

Graphics Processing Units (GPUs) are power-efficient and high-memory-bandwidth processors that can exploit parallelism in computationally demanding general purpose applications [23]. GPUs, due to the SIMD-like (Single Instruction Multiple Data) architecture, have proved to be extremely effective at accelerating operations on traditional vector- or matrix-based data structures, which exhibit massive data parallelism, regular memory access patterns, few synchronizations and a straightforward mapping to parallel hardware.

However, many application domains employ applications that create, traverse, and update irregular data structures such as trees, graphs, and priority queues. Recent low-level hardwired graph primitive implementations for breath-first search in b40c [1], graph connectivity [2], betweenness centrality [4, 24], and single-source shortest path [25], have demonstrated the strong computational power of modern GPUs in bandwidth-hungry graph analytics. Unfortunately, mapping irregular graph primitives to parallel hardware is non-trivial due to the data-dependent control flow and unpredictable memory access patterns [26–30].

For more general real-world graph analytics, developers need a high-level GPU programming interface to implement various types of complex graph applications on the GPU without sacrificing much performance. Programming models are key design choices that impact both the expressiveness and performance of graph analytic frameworks. Most GPU+graph programming models today mirror CPU programming models: for instance, Medusa [31] uses Pregel's message passing model, and VertexAPI2 [32], MapGraph [33] and CuSha [34] use and modify PowerGraph's Gather-Apply-Scatter (GAS) programming model. Our more recent GPU high-level graph framework, Gunrock [30], uses a GPU-specific data-centric model focused on operations on a subset of vertices and/or edges. Many frameworks have been extended to multi-GPU platform [31, 35] to deliver high-performance and scalability while maintaining programmability and compatibility. Hybrid CPU+GPU graph analytic frameworks, such as Totem [36], also gained many attentions.

Efficient and scalable mechanisms to schedule workloads on parallel computation resources are imperative. The unpredictable control flows and memory divergence on GPU introduced by irregular graph topologies need sophisticated strategies to ensure efficiency. And yet, unlike CPU graph frameworks, little is known about the behavior of high-level GPU frameworks across a broad set of graph applications. This thesis is trying to understand one problem: what kind of frameworks, libraries and application programming interfaces can be used in the development graph primitives and what are their advantages and disadvantages? This leads to the next question: what are critical factors that impact the efficiency of graph processing on the GPU? Are they influenced by graph topology? Or by programming model? How can we build more programmable and efficient graph analytics framework? What is the "smart" way of thinking about graph analytic problems on parallel architecture? Evaluating the performance is essential for tuning, optimization and framework selection. Single GPU workload is critical as a basic building block of future graph processing frameworks on multiple GPUs and GPU clusters. A thorough investigation of single GPU graph framework performance can provide insights and potential tuning and optimizations to accelerate a class of irregular applications with further benefit to future graph analytic frameworks on multi-GPUs and GPU clusters.

1.3 Previous Work

Developing and evaluating graph primitives on GPU is a hot recent topic. Xu et al. [28] studied 12 graph applications in order to identify bottlenecks that limit GPU performance. They show that graph applications tend to need frequent kernel invocations and make ineffective use of caches compared to non-graph applications. Pannotia [27] is a suite of several specific GPGPU graph applications implemented in OpenCL used to characterize the low-level behavior of SIMD architectures, including cache hit ratios, execution time breakdown, speedups over CPU version execution, and Single Instruction Multiple Thread (SIMT) lane utilization. O'Neil et al. [29] presented the first simulator-based characterization, which focused on the issue of underutilized execution cycles due to irregular graph codes and addressed the effectiveness of graph-specific optimizations. Burtscher et al. [26] defined two measures of irregularity—Control-Flow Irregularity (CFI) and Memory-Access Irregularity (MAI)—to evaluate irregular GPU kernels. Their contributions can be summarized as: a) irregularity varies across different applications and datasets; b) common performance bottlenecks include underutilized execution cycles, branch divergence, load imbalance, synchronization overhead, memory coalescing, L2/DRAM latency, and DRAM bandwidth; c) improvements in memory locality/coalescing and fine-grained load balancing can improve performance. Yang and Chien [37] studied different ensembles of parallel graph computations, and concluded that graph computation behaviors form an extremely broad space. Beamer et al. [38] using diverse workload to demonstrate that most of workload fails to fully utilize memory bandwidth.

Previous graph processing workload characterizations are dominated by architectural-level behavior and simulation-based analysis. However, the high-level abstractions for graph analytics on GPU-based frameworks, and their impact on graph workloads, have not been investigated in detail. What remains unclear is how to map these low-level optimizations and performance bottlenecks to different high-level design choices in order to find best programming model and a set of general principles for computing graph analytics on the GPU. Previous characterization work is also limited to individual graph primitives implemented on their own rather than examining state-of-the-art general-purpose graph analytic frameworks. Unlike previous benchmarking efforts, we focus more on the performance and characteristics of high-level programming models for graph analytics on GPUs.

1.4 Main Contributions

This thesis tries to achieve the above goal. Our contributions are as follows:

- We identify main factors that are critical to efficient high-level GPU graph analytic frameworks and present a performance characterization of three existing frameworks on the GPU—VertexAPI2, MapGraph, and Gunrock—by exploring the implementation space of different categories of graph primitives and their sensitivity to diverse topologies.

- We present a detailed experimental evaluation of topology sensitivity and identify the key properties of graphs and their impact on the performance of modern GPU graph analytic frameworks.

- We investigate the effect of design choices and primitives used by the three high-level GPU graph analytic frameworks above and key design aspects for more efficient graph analytics on the GPU.

1.5 Thesis Organization

The rest of the thesis is organized as follows: *Chapter 2* provides the necessary background and motivation on graph processing using high-level frameworks (Section 2.1) and reviews several common graph primitives used in this work as case studies in Section 2.2. In Section 2.3, we introduce two existing popular graph programming models and their implementations on GPU. *Chapter 3* first identifies several important factors that impact graph analytics performance on GPUs in Section 3.1. Then we discuss and define the methodology and performance metric used to investigate the programming models (Section 3.2). Section 3.3 specifies the testing environment in this study. Detailed empirical studies are provided in *Chapter 4*. Finally, *Chapter 5* discusses some efforts that potentially can be made to further improve graph processing abstractions. The main body of the thesis is followed by *bibliography* which contain useful references information.

Chapter 2

Preliminaries and Abstractions

2.1 Preliminaries

Let an unweighted graph be $G = (V, E)$ where V is the set of vertices and E is the set of edges in the graph, where $E \subseteq (V \times V)$. We denote a weighted graph by $G = (V, E, w)$, where w maps an edge to a real value associated with E. We denote number of vertices by $|V|$ and similarly, $|E|$ for number of edges in the graph. A path from a vertex u to a vertex v is any sequence of edges originating from u and terminating at v. An *undirected* (*symmetric*) graph implies that for all pairs of $(u, v) \in V : (u, v) \in E$ and $(v, u) \in E$. Otherwise, it is a *directed* graph. In graph processing, a *vertex frontier* represents a subset of vertices $U \in V$ and similarly an *edge frontier* represents a subset of edges $I \in E$.

Representation The *adjacency list* format for graph representation stores for each vertex an array of indices of other vertices that it has an edge to as well as the vertex's degree. Compressed Sparse Row (CSR) is a concise representation of adjacency list that store arrays consecutively in memory as shown in Fig 2.1. CSR is a good fit for its space-efficiency and its ability to

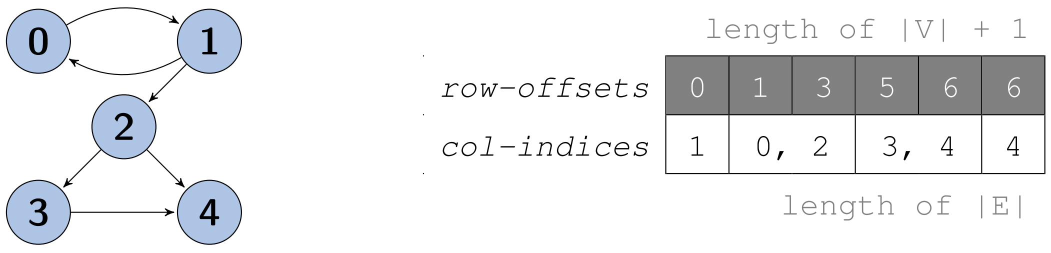


Figure 2.1: Compressed Sparse Row (CSR) graph representation

strip out adjacency lists and find offsets using parallel-friendly primitives. CSR contains a row pointer array (*row-offsets*) that stores the start of each adjacency list and a column indices array (*col-indices*) that contains the concatenation of the neighbor list of each vertex. Compress Sparse Column (CSC) is similar to CSR except it stores column pointers and row indices. This representation is space-efficient which minimizes memory use to $\mathcal{O}(|V|+|E|)$.

2.2 Graph Primitives

We observe that graph primitives can be divided into two broad groups. First, *traversal-based* primitives start from a subset of vertices in the graph, and systematically explore and/or update their neighbors until all reachable vertices have been touched. Note that only a subset of vertices is typically active at any point in the computation. In contrast, most or all vertices in a *dense-computation-based* primitive are active in every stage of the computation.

Breath-First Search (BFS): BFS is a visiting strategy for vertices of a graph that systematically explores the connected vertices of a graph staring from a given source vertex $v_{source} \in V$ such that all vertices are visited in the order of hop-distance from the given source vertex. BFS is a common building block for more sophisticated graph primitives, such as eccentricity estimation [39], betweenness centrality, and connected component. BFS is representative of a class of irregular and data-dependent parallel computations [1]. Other traversal-based primitives include Betweenness Centrality (BC), a quantitative measure to capture the effect of important vertices in a graph, and shortest path finding, which calculates the shortest distance between source and all other vertices in a weighted graph, and many others. The work complexity of BFS is $\mathcal{O}(|V|+|E|)$.

Single-Source Shortest Path (SSSP): For a weighted graph $G=(V,E,w)$, beginning with the source vertex $v_{source} \in V$, SSSP calculates the shortest distance (minimum sum of weights) between the *source* and any reachable vertex in the graph. All vertices start with an infinite (∞) distance except for source vertex, then the distances are updated by examining the neighbors in each iteration to find the one with minimum cost after including the path to that neighbor. The above primitives are generally considered traversal-based.

Connected Component (CC): For an undirected graph, CC finds each subset of vertices

$C \subseteq V$ in the graph such that each vertex in C can reach every other vertex in C, and no path exists between vertices in C and vertices in $V \backslash C$. The simple approach to calculate graph connectivity is to use BFS traversal, or it can be implemented using label propagation [2]. Depending on its implementation, connected component can be in either category.

PageRank: Link analysis and ranking primitives such as PageRank calculate the relative importance of vertices and are dominated by vertex-centric updates; PageRank is an example of a dense-computation-based primitive as opposed to traversal-based primitives that are container-centric focusing on particular frontier of vertices/edges at any point of computation. The PageRank score for a web page A can be calculated by iterating Equation 2.1, where *delta* is a damping factor, *degree* is the number of out-going links from web page x, and the set $x \in A$ denotes the set of all web pages x that have a link to A.

$$rank_A = (1 - delta) + delta \times \sum_{x \in A} \frac{rank_x}{degree_x} \tag{2.1}$$

In this work, we use four common primitives—BFS, SSSP, CC and PageRank—as case studies to benchmark the performance of the three mainstream programmable graph analytic frameworks on the GPU. BFS represents a simpler workload per edge while SSSP includes more expensive computations and complicated traversal paths. CC and PageRank involve dense computations with different per vertex/edge workloads. The behavior of these primitives is diverse, covering both memory- and computation-bound behaviors, and together reflects a broadly typical workload for general graph analytics.

2.3 GPU Graph Abstractions

Many graph primitives can be expressed as several inherently parallel computation stages interspersed with synchronizations. Existing programmable frameworks on the GPU [30, 32, 33, 40] all employ a Bulk-Synchronous Parallel (BSP) programming model in which users express a graph analytic program a series of consecutive *"super-steps"*, separated by global barriers at the end of each super-step, where each super-step exhibits ample data parallelism and can be run efficiently on a data-parallel GPU. The operations within each super-step may include per-vertex and/or per-edge computations that run on all or a subset of vertices/edges in a graph, for instance, reading and/or updating each vertex's/edge's own associated data or that of its neighbors. A super-step may instead traverse the graph and choose a new subset of active vertices or edges, known as *"frontier"*, during or after graph computations. Thus the data parallelism within a super- step is typically data parallelism over all vertices or edges in the frontier. The programs themselves are generally iterative, convergent processes that run super-steps that update vertex/edge values repeatedly until they reach a termination condition (*"convergence"*).

The major differences of most high-level graph analytic frameworks lie in two aspects: 1) how a super-step is defined to update vertices and/or edges in the current frontier and 2) how to determine and generate a new frontier for the next super-step. In this work, we study VertexAPI2 [32], an implementation strictly following the Gather-Apply-Scatter (GAS) model; MapGraph [33], a modified version of the GAS model; and Gunrock [30], our data-centric abstraction that focuses on manipulations of frontiers. All three frameworks run bulk-synchronous super-steps on a frontier of active vertices/edges until convergence. **Notation**: In performance figures of this thesis, we use VA for VertexAPI2, MapGraph for MapGraph, and GR for Gunrock.

2.3.1 Gather-Apply-Scatter Model

The Gather-Apply-Scatter (GAS) approach was originally developed for distributed environments [18, 20]. The GAS model decomposes a vertex program into three conceptual phases: *gather*, *apply*, and *scatter*. The gather phase accumulates information about adjacent vertices and edges of each active vertex through a generalized binary operation over its neighbor list, for instance, binary plus ($\oplus$) can be used in gather to accumulates the sum of the neighbor list. The apply phase computes the accumulated value, the output of the gather phase, to the active vertex as a new vertex attribute. And during the scatter phase, a predicate is evaluated on all adjacent outgoing-edges and corresponding vertices. A vertex carries two states: *active* and *inactive*. Both VertexAPI2 and MapGraph broadly follow the GAS model, but with important differences. VertexAPI2 disables the scatter phase as none of the four primitives in its current implementation can be expressed without push-updates; instead, after gather and apply, VertexAPI2 writes predicate values to a $|V|$-length Boolean flag, then invokes an *activate* phase

Listing 2.1: MapGraph

```
Program::Initialize(); // Initialization
//  Repeat until the frontier is empty
while (frontier_size > 0) {
  gather();   // Doing nothing
  apply();    // Doing nothing
  expand();   // Expanding neighbors
  contract(); // Get new frontier
}
Problem.ExtractResults(); // Get Results
```

Listing 2.2: VetexAPI2

```
engine.setActive(); // Initialization
//  Repeat until no active vertex exist
while (engine.countActive()) {
  engine.gatherApply(); // Update labels
  engine.scatterActivate(); // Get new
  engine.nextIter();      // active list
  setIterationCount();    // Count level
}
engine.getResults(); // Get results
```

Listing 2.3: Gunrock

```
BFSProblem::Init(); // Initialization
//  Repeat until the frontier is empty
while (frontier_queue_length > 0) {
  // Get neighbors and update labels
  BFSEnactor::gunrock::oprtr::advance();
  // Cenerate new vertex frontier
  BFSEnactor::gunrock::oprtr::filter();
}
BFSProblem::Extract(); // Get result
```

Figure 2.2: Code snapshots for three different GPU graph analytic frameworks (BFS)

to scan and compact vertices associated with true flags to create frontiers for the next super-step. MapGraph instead decomposes the scatter into two phases: *expand*, to generate edge frontiers and *contract*, to eliminate duplicates in new frontiers that arise due to simultaneous discovery. To improve flexibility, the gather and scatter phases in MapGraph support in-edges, out-edges, or both.

2.3.2 Data-Centric Model

Rather than focusing on expressing sequential super-steps of computation on vertices, Gunrock's abstraction focuses on manipulations of the frontier of vertices and/or edges that are actively participating in the computation. Gunrock supports three ways to manipulate the current frontier: *advance* generates a new frontier by visiting the neighbors of the current vertex frontier; *filter* generates a new frontier by choosing a subset of the current frontier based on programmer-specified criteria; and *compute* executes an operation on all elements in the current frontier in parallel. Gunrock's advance and filter dynamically choose optimization strategies during runtime depending on graph topology. In order to reduce the number of kernel invocations and enhance producer-consumer locality, Gunrock's compute steps are expressed as device functions that can be fused into advance and filter kernels at compile time. Figure 2.2 shows how we can express BFS using operators in three different frameworks.

Chapter 3

Methodology

3.1 Critical Factors for Efficiency

In this section, we identify and discuss important factors that are critical to fast GPU graph analytics. Investigating these issues and evaluating design choices are necessary for building high-level graph analytic frameworks.

3.1.1 Building Block Operators

Building block operators are vital for graph analytic frameworks. Being efficient can mean: 1) using these operators flexibly yields high-performance outcomes; 2) the operators themselves are implemented efficiently. The former affects how graph primitives are defined and expressed, which result in abstraction-level and performance differences; the latter impacts workload distribution and load balancing, as well as memory access patterns across the GPU. As a result, efficient building block graph operators are invariably tied to performance. Example operators for the three frameworks in Section 2.3 include gather, apply, expand, contract, activate, advance, and filter.

3.1.2 Workload Distribution

Workload distribution and load balancing are crucial issues for performance; previous work has observed that these operations are dependent on graph structure [27, 28]. Hardwired graph primitive implementations [1, 2, 4, 24, 25] have prioritized efficient (and primitive-customized) implementations of these operations, thus to be competitive, high-level programmable frameworks must offer high-performance but high-level strategies to address them. While some operators are simple to parallelize on the GPU—GAS's apply is perfectly data-parallel; Gunrock's filter is more complex but still straightforward—others are more complex and irregular. Traversing neighbor lists, gather, and scatter, for instance, may each have a different workload for every vertex, so regularizing workloads into a data-parallel programming model is critical for efficiency.

3.1.3 Synchronization and Data Movement

Synchronizations and data movements are limiting factors under widely-adopted BSP models. While the BSP model of computation is a good fit a GPU, the cost of its barrier synchronization between super-steps remains expensive. Asynchronous models that may reduce this cost are only just beginning to appear on the GPU [41]. Beyond BSP synchronization, any other global synchronizations within BSP super-steps are also costly. Expert programmers often minimize global synchronizations and unnecessary data movement across device kernels. How do the high-level programming models we consider here, and the implementations of their operations, impact performance?

3.1.4 Memory Behavior

Memory access patterns and usage are also main limiting factors for GPU performance. In graph primitives, memory access patterns are both irregular and data-dependent, as noted by previous work [26]. Thus efficient utilization, better access patterns for both global and shared memory, fewer costly atomic operations, and less warp-level divergence contribute to superior overall performance. Because GPU on-board memory capacity is limited, efficient global memory usage is particularly important for addressing large-scale graph workloads.

3.2 Graph Topology

The performance of graph primitives is highly topology-dependent. Most GPU graph primitives have much higher traversal rates on scale-free graphs then on road networks as we describe in more detail in Section 4.1. We explore how graph topology impacts the overall graph analytics performance using the following metrics. The *eccentricity* $\varepsilon(v)$ is the maximum distance between a vertex v and any other vertex in the graph. The *radius* of a graph is the minimum graph eccentricity $r = \min\ \varepsilon(v)$ and in contrast, the *diameter* of a graph is the maximum length of all paths between any pair of vertices $d = \max\ \varepsilon(v)$. The magnitude of *algebraic connectivity* reflects the well-connectedness of the overall graph. For traversal-based graph primitives, traversal depth (or search depth, number of iterations) is directly proportional to the eccentricity and connectivity. The *vertex degree* implies the number of edges connected to a vertex. The average number of degrees of a graph and its degree distribution determine the amount of parallelism; and unbalanced degree distribution can significantly impact the load balancing during graph traversal.

Real-world graph topologies usually fall into two categories: the first contains small eccentricity graphs with highly-skewed scale-free degree distributions, which results in a subset of few extremely high-degree vertices; the second has large diameters with evenly-distributed degrees. In Chapter 4, we choose diverse datasets that encompass both categories, and also generate several synthesized [42, 43] graphs whose eccentricity and diameter values span from very small to very large.

Table 3.1 summarizes our benchmark suite and Table 3.2 (Line 1) shows the degree distribution of four graphs from each group. Social networks commonly have scale-free vertex degree distributions and small diameter (thus small depth). Synthetic Kronecker datasets (`kron_g500-logn17` $\sim$ `kron_g500-logn21`) are similar to social graphs with the majority of the vertices belonging to only several levels of BFS. Delaunay datasets (`delaunay_n17` $\sim$ `delaunay_n21`) are Delaunay triangulations of random points in the plane have extremely small out-degrees. Road networks and open street maps (OSMs) are two types of real-world road networks with most (87.1%) vertices having an directed out-degree below 4. Table 3.2 (Line 2) depicts the vertex frontier and edge frontier sizes as a function of iteration running graph traversals, other graphs in the same group follow similar patterns. Road networks and Delaunay meshes usually have comparable vertex and edge frontier sizes; however, for social networks and Kronecker datasets, edge frontiers are enormous compared to the vertex frontiers. We group the datasets above into four different groups: road, delaunay, social, and kron. Graphs in a group share similar topology and thus similar behavior. Overall, these datasets cover a wide range of graph topologies to help us characterize and understand the

Dataset		Graph Scale		Vertex Degree		Search Depth		
Group	Name	Vertices	Edges	Max	Average	Min	Max	Average
road	roadNet-CA	1.97M	5.53M	12	2.81	3	846	657
road	asia_osm	12.0M	25.4M	9	2.13	26,516	45,724	36,077
road	road_central	14.1M	33.9M	8	2.41	2,950	5,382	4,208
road	road_usa	23.9M	57.7M	9	2.41	4,317	8,307	6,155
road	europe_osm	51.0M	108M	13	2.12	15,402	28,326	19,338
dely	delaunay_n17	131k	393k	17	6.00	143	167	155
dely	delaunay_n18	262k	786k	21	6.00	197	228	214
dely	delaunay_n19	524k	1.57M	21	6.00	273	319	295
dely	delaunay_n20	1.05M	3.15M	23	6.00	379	445	413
dely	delaunay_n21	2.10M	6.29M	23	6.00	529	620	571
social	amazon-2008	735k	7.05M	1,077	9.58	18	25	20.9
social	hollywood-2009	1.14M	113M	11,467	98.9	0	11	8.47
social	tweets	1.85M	5.75M	61,038	3.12	2	22	16.9
social	soc-orkut	3.00M	213M	27,466	71.0	7	9	8.20
social	soc-LiveJournal1	4.85M	85.7M	20,333	17.7	12	20	13.9
kron	kron_g500-logn17	131k	10.1M	29,935	78.0	0	7	4.40
kron	kron_g500-logn18	262k	21.0M	49,162	80.7	0	7	4.59
kron	kron_g500-logn19	524k	43.2M	80,674	83.1	0	7	5.09
kron	kron_g500-logn20	1.05M	88.6M	131,503	85.1	0	7	4.73
kron	kron_g500-logn21	2.10M	181M	213,904	86.8	0	7	4.60

Table 3.1: Our suite of benchmark datasets, all converted to symmetric graphs. Degree indicates the average vertex degree for all vertices and depth is the average search depth randomly sampled over at least 1000 BFS runs. We group the datasets above into four different groups: road, delaunay, social, and kron. Datasets in the same group share similar structure and thus similar behavior

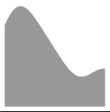
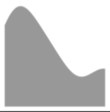
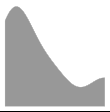
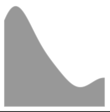
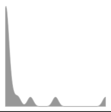
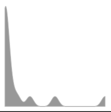
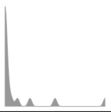
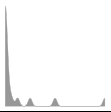
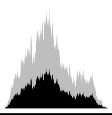
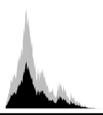
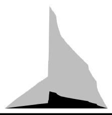
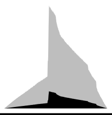
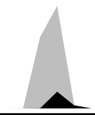
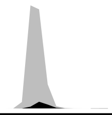
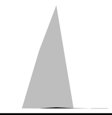
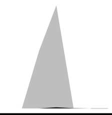
Dataset	road_usa	europe_osm	delaunay-n17	delaunay-n21	soc-orkut	soc-liveJournal1	kron_g500-logn17	kron_g500-logn21
Distribution								
FrontierSize								

Table 3.2: Degree distribution and BFS frontier size: vertex (dark) and edge (light) frontier sizes as a function of iteration running BFS on two graphs in each group. Note that road networks and Delaunay meshes usually have comparable vertex and edge frontier sizes; however, for social and Kronecker datasets, edge frontiers are enormous compared to the vertex frontiers

performance and effectiveness of different graph analytic frameworks. Most of our real-world graphs are taken from the 10th DIMACS Challenge, the University of Florida Sparse Matrix Collection [44]; we also use synthetic graph regular random generators [42] and GTgraph [43] (`R-MAT`) with parameters a = 0.57, b = c = 0.19 (choosing b = c for symmetry), d = 0.05. To provide a weighted input for the SSSP primitives, we associate a random integer weight in the range [1, 128) to each edge.

3.3 Experiment Environment

We empirically evaluate the effectiveness of graph primitives (Section 2.2), expressed in the GAS implementations of VertexAPI v2, MapGraph v0.3.3 and our data-centric implementation of Gunrock [1] discussed in Section 2, on our benchmark suite. We first present the overall performance comparison of three frameworks, then dig into the detailed impacts of input datasets, framework abstractions, and various optimizations. This section summarize the machines and compilers used for experimental evaluation throughout this thesis.

Hardware: All experiments ran on a Linux workstation ("mario") with 2 $\times$ 3.50 GHz Intel(R) 4-core E5-2637 v2 Xeon(R) CPUs. The machine has a total of 512 GB of DDR3 main memory. In this thesis, we use a Tesla K40c NVIDIA GPU from the Kepler generation with 12 GB on-board GDDR memory and compute capability 3.5. Tesla K40c has 15 vector processors, termed streaming multiprocessors (SMX), each containing 192 parallel processing cores, called streaming processors (SP). NVIDIA GPUs use the Single Instruction Multiple

[1]Using git commit 3f1a98a3ec64ee72cee43fbaea68f3e1f553703c on May 8, 2015

Model	Tesla K40c	Architecture	Kepler GK110B
Compute capability	3.5	SMXs (cores per SMX)	15,192
Warp size	32	Max threads per CTA	1,024
Max threads per SMX	2,048	Shmem per CTA (KB)	48
L2 cache (KB)	1,536	Total global memory (MB)	12,288
Peak off-chip BW (GB/s)	288	Peak GFLOP/s (DP FMA)	1,430

Table 3.3: NVIDIA Tesla K40c GPU used in experiments

Thread (SIMT) programming model. GPU device programs, *kernels*, run on a large number of parallel threads to achieve massive data parallelism. Each set of 32 or 64 single threads forms a 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 called *blocks*, also called Cooperative Thread Array (CTA), whose threads can communicate through a pool of on-chip shared memory. K40c has 48 KB on-chip shared memory per streaming multiprocessor (SMX). All SMXs share an off-chip global DRAM. The cards were connected to the host (CPU) via two PCIe I/O hubs. A program may consist of one or more kernels, each consisting of one or more cooperative thread arrays (CTAs), and each CTA consists of multiple warps. Details are summarized in Table 3.3.

Compilers: The parallel GPU CUDA programs were compiled with NVCC compiler (version 6.5.12). The sequential C/C++ code was compiled using GCC (Ubuntu 4.8.1-2ubuntu1 12.04) 4.8.1 with the -O3 optimization level. In this thesis, we aim to focus on an abstraction-level understanding of the frameworks centered on their GPU implementations; all results ignore transfer time (disk-to-memory and host-to-device). We use NVIDIA visual profiler (nvprof) to collect some of our characterization results.

Chapter 4

Evaluation and Analysis

We begin our evaluation by looking at primitive performance on a variety of graph types and analyzing how the datasets influence each framework's performance and information propagation. Then to better understand the overall performance, we focus on the following details of the abstraction and implementations: load balancing and workload distribution, synchronization, and memory consumption and behavior.

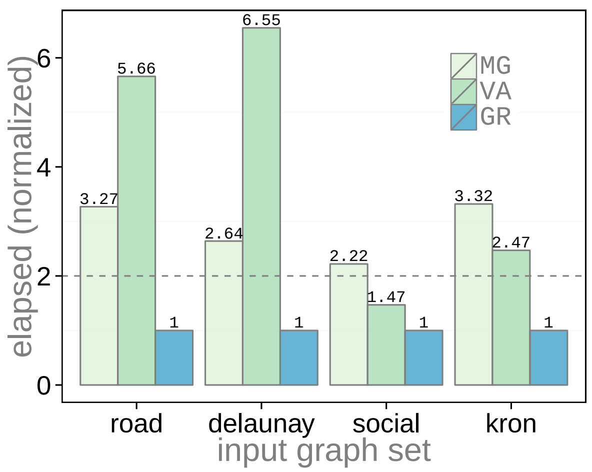
4.1 Overall Performance

Figure 4.1 contains the normalized runtime for four graph primitives across all datasets on three frameworks. We observe that: 1) the runtime ratios of a primitive evaluated on the three frameworks can heavily differ simply because of the topology of the input graph; 2) for traversal- based primitives, all frameworks have much higher traversal rates (lower elapsed runtime) on scale-free graphs then on road networks. In the rest of this section, we identify and characterize abstraction-level trade-offs and investigate the reasons and challenges for graph analytics behind these observed differences in overall performance.

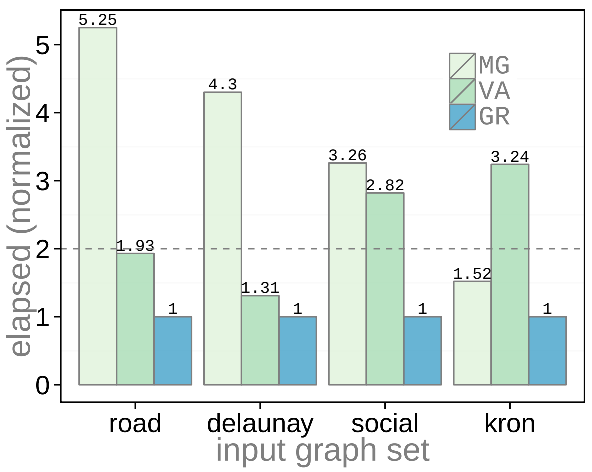
4.2 Input Dataset Impacts

4.2.1 Vertex/Edge Frontier Size

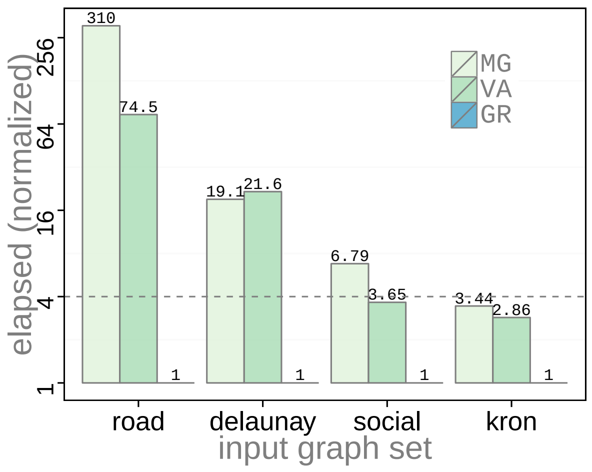
Ample parallelism is critical for GPU computing. To make full use of the GPU, programmers must supply sufficient workload to the GPU to keep its many functional units occupied with enough active threads to hide memory latency. The NVIDIA Tesla K40c we use here has



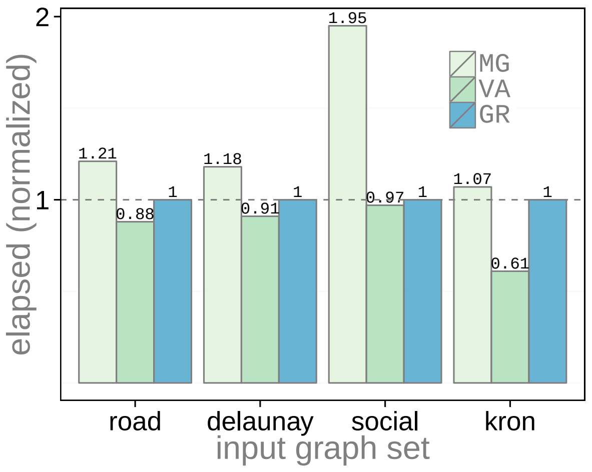
(a) Breadth-First Search (BFS)



(b) Single-Source Shortest Path (SSSP)



(c) Connected Component (CC)



(d) PageRank

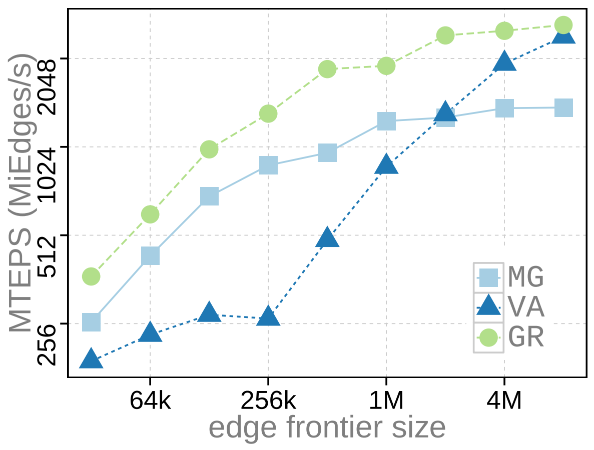
Figure 4.1: Overall runtime on four primitives and four groups of input datasets (normalized to Gunrock's runtime. All Gunrock's BFS results disable direction-optimizing because neither MapGraph nor VertexAPI2 support this optimization). We report the execution time of first 20 super-steps for PageRank

15 multiprocessors, each of which can support 2048 simultaneous threads; thus ~30k active threads are the bare minimum to keep the GPU fully occupied, and it is likely that many times that number are required in practice.

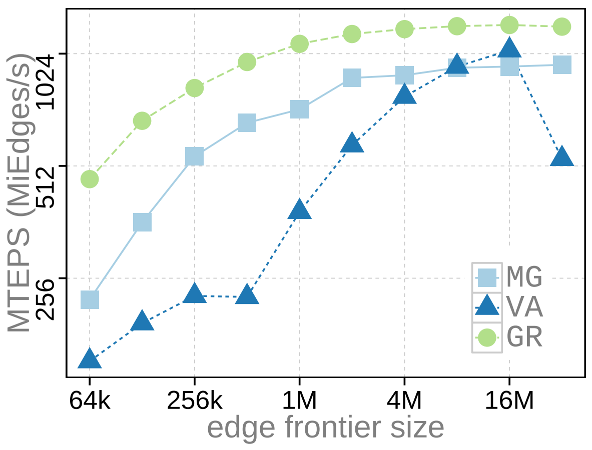
We begin with a best-case scenario to evaluate the graph traversal: two traversal-based primitives, BFS and SSSP on a synthetic perfectly load-balanced graph with no redundant work (no concurrent child vertex discovery). Such a graph should allow a graph framework to achieve its peak throughput. Figure 4.2 (a) illustrates the throughput in Millions of Traversed Edges Per Second (MTEPS, higher is better) as a function of the edge frontier size at each BFS level during the traversal. Generally, throughput increases as the edge frontier size increases. This behavior is expected—larger edge frontiers means more parallelism—but for any of the frameworks, the GPU does not reach its maximum throughput until it has millions of edges in its input frontier. (The number of frontier edges for maximum throughput corresponds to on the order of 100 edges processed per hardware thread.) This requirement is a good match for scale-free graphs (like social networks or kron), but means that less dense, low-degree and high- diameter graphs like road networks will be unlikely to achieve maximum throughput on BSP-based GPU graph frameworks.

VertexAPI2's saturation point exceeds the largest synthetic frontier size we used. We attribute such scalability and performance boost for VertexAPI2 to their switching of parallelism strategy beyond a certain frontier size. These behavioral patterns suggest that ample workload is critical for high throughput, and thus scale-free graphs will show the best results in general across all frameworks. Between frameworks, the BFS topology preference observations in Section 4.1 are confirmed by our results here: MapGraph and Gunrock performs more advantageously on long- diameter road networks and meshes while VertexAPI2 benefits most from social networks and scale-free kron graphs.

Let's now turn to two different real-world datasets: a high-diameter road networks, `roadNet-CA`, and a scale-free social graph, `hollywood-2009`. We show MTEPS for these graphs in Figure 4.3 (`RoadNet-CA` is sampled). Peak MTEPS differ by several orders of magnitude, which we can directly explain by looking at actual edge frontier sizes: `roadNet-CA`'s range from 3 to 17,780 ($\approx 2^{14}$), while `hollywood-2009`'s peaks at 58.1M



(a) Breadth-First Search (BFS)



(b) Single-Source Shortest Path (SSSP)

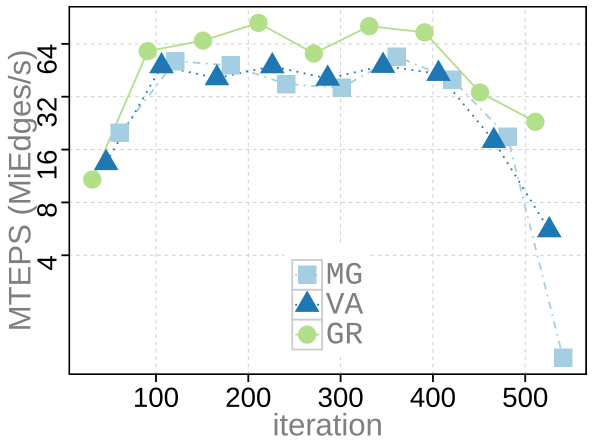
Figure 4.2: Millions of Traversed Edges Per Second (MTEPS) vs. edge frontier size: measured on synthetic trees with no concurrent child discovery (redundant work)

(between 2^{25} and 2^{26}), which is substantially larger than the saturation frontier size we observed in Figure 4.2. The performance patterns are consistent with the previous key findings; VertexAPI2 achieves peak performance for two super-steps of the largest edge frontiers, which explains its behavior on social networks and kron datasets.

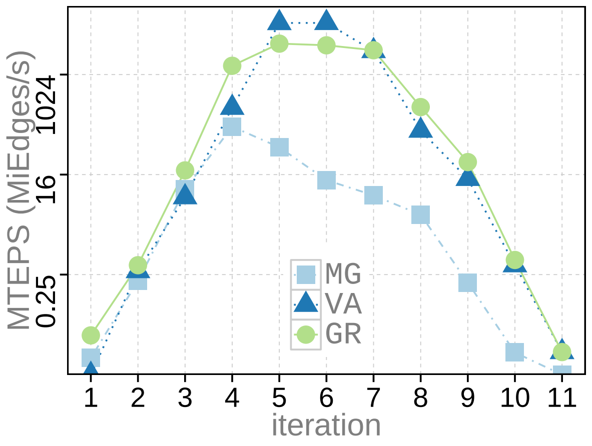
BFS and SSSP are traversal-based and thus only have a subset of vertices or edges in a graph active in a frontier at any given time. We see that when that fraction is small, we do not achieve peak performance. In contrast, a dense-computation-based primitive like PageRank has all vertices active in each super-step before convergence. Such a primitive will have a large frontier on every super-step and be an excellent fit for a GPU; in practice, all three frameworks perform similarly well on PageRank.

4.2.2 Per-Vertex Workload

Another important characteristic of graphs is the average degree (average neighbor list size), which we can use to quantify per-vertex workload. Figure 4.4 shows the performance impact of average degree of graph traversal with a set of synthetic regular random geometry graphs [42], and scale-free `R-MAT` graphs [43]), all with the same scale, and thus same number of vertices, and only differing in average degree. All frameworks demonstrate better performance with increasing degree; more work per vertex leads to higher throughput. Smaller average degree

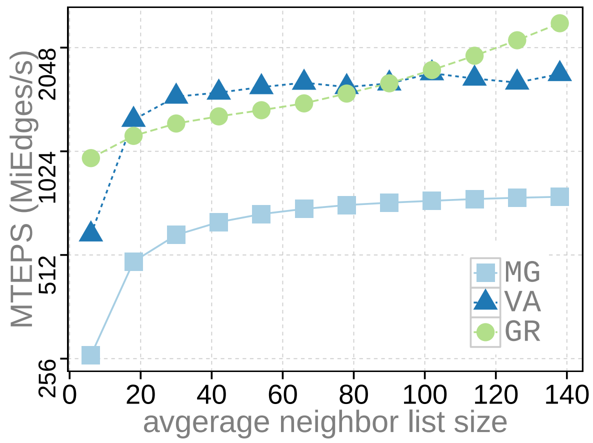


(a) MTEPS for each level: `roadNet-CA`

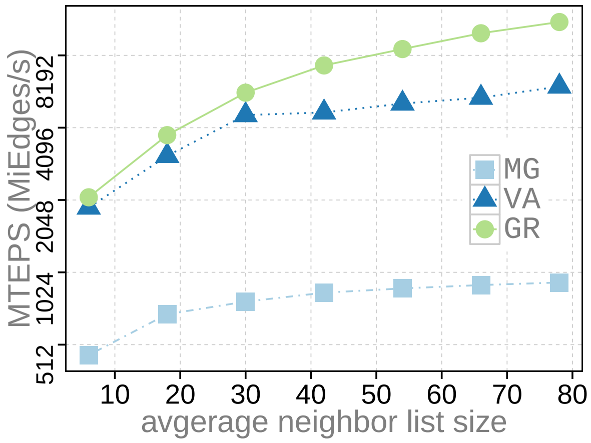


(b) MTEPS for each level: `hollywood-2009`

Figure 4.3: Millions of Traversed Edges Per Second (MTEPS) for each super-step of two real datasets: `roadNet-CA` and `hollywood-2009` (`roadNet-CA` is sampled)



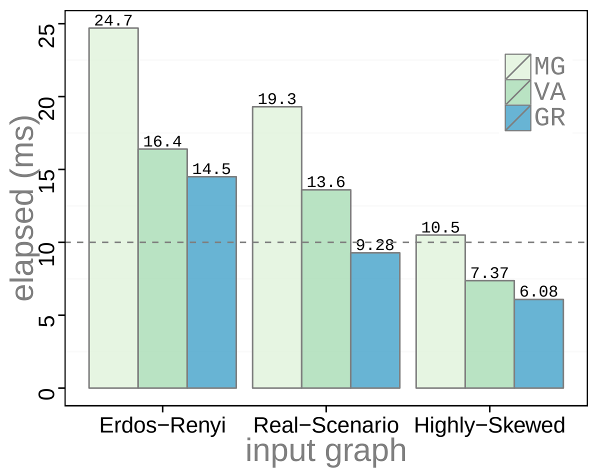
(a) Random Regular Graphs



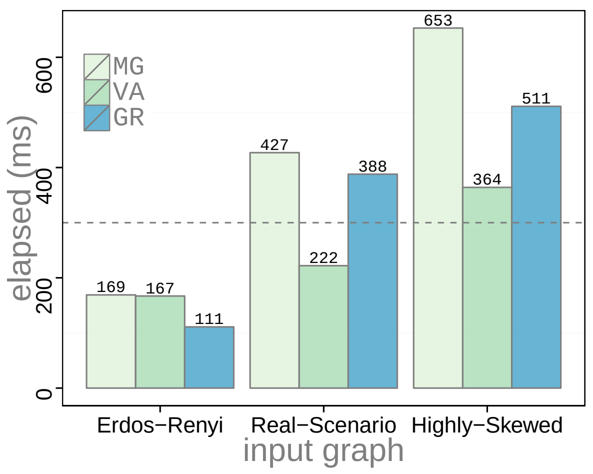
(b) Scale-free `R-MAT` Graphs

Figure 4.4: Impact of average degrees on performance (measured in MTEPS): each graph in the same group has fixed vertex count and varying vertex average degrees

graphs (e.g., road networks and OSMs) are limited by available parallelism. Larger average (over 20 $\sim$ 30) graphs demonstrate notably higher MTEPS for all frameworks. For graphs with small vertex degrees or small total number of vertices/edges, there are problems to feed many parallel threads. In practice, this larger degree is necessary to provide enough parallelism to keep the GPU fully occupied.



(a) Traversal-based: BFS



(b) Computation-based: PageRank

Figure 4.5: Impact of skewness on performance (measured in runtime): all three graphs have the same vertex count and edge count with different topology generated using `R-MAT` generator

4.2.3 Distribution and Skew-Freeness

Finally, scale-free degree distribution and their skewness also impact performance. We can broaden our analysis by varying the skewness through varying parameters of the `R-MAT` graph generator to create not only social-network-like graphs but also graphs with other behavior as well. We set the ratio of the parameters a and b, c (choosing b = c for symmetry) to 8 (highly-skewed scale-free graph), 3 (closet to many real world scenarios) and 1 (Erdös Rényi model). The result is three synthetic graphs with the same vertex count and edge count but different skewness. Figure 4.5 shows our runtime results for BFS graph traversal and PageRank on these three graphs.

On the traversal-based BFS and SSSP, we see lower runtime as skewness increases and eccentricity decreases; runtime is most correlated to the number of super-steps (iterations) to traverse the entire graph, and the highly-skewed graphs have the smallest diameter. On the other hand, for computation-based PageRank, we see the opposite runtime behavior: the high amount of skew yields noteworthy load-balancing challenges that hurt the overall runtime compared to less-skew graphs.

In Summary: on large and dense graphs, performance is generally better than on sparse graphs. The GPU shines when given large frontiers with large and uniformly-sized vertex frontiers, and when diameters are low. It struggles with small frontier sizes, with small vertex degrees and load imbalances within the frontier, and with more synchronizations caused by high diameters. Dense, scale-free, low-diameter graphs like social networks are particularly well suited for the GPU; road networks are a poor fit to the frameworks we study here, because they do not expose enough work to saturate the GPU. In general, traversal-based primitives will be more affected by topology than computation-based primitives, because they operate on a smaller subset of the graphs and hence have less parallel workload to do per step. Now we turn away from *how* our graph frameworks perform on different topologies to the underlying reasons *why* they perform that way.

4.3 Load Balancing Impacts

For graph analytics, the amount of parallelism is dynamic, time-varying, workload-dependent, and hard to predict [27, 28]. The programmable frameworks we study here [30, 32, 33] encapsulate their solution to this problem in their operators, which must capture this parallelism at runtime. Thus, the design choices of Gunrock's advance traversal operation and GAS's gather and scatter operations can appreciably impact performance.

4.3.1 Load Balancing Strategies

The graph analytic frameworks we study use three distinct techniques to achieve parallel workload mapping at runtime:

- When the frontier is so small that there is no way to fully utilize the GPU, load balancing is not a major concern. The simple strategy is thus the popular one: a per-thread neighbor list expansion (PT), where an entire vertex's neighbor list is mapped to a single thread. However, as frontiers get larger and neighbor list sizes differ by several orders of magnitude, PT's load-balancing behavior becomes unacceptably bad.

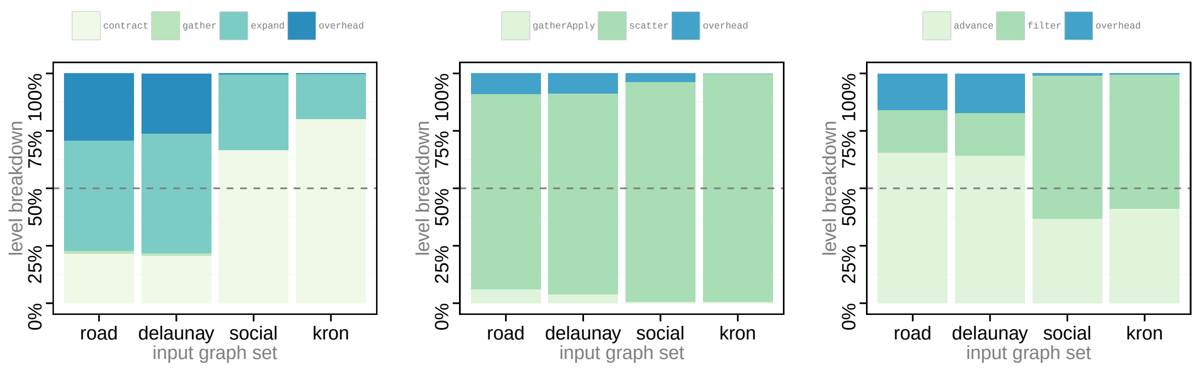
- One alternative is dynamic workload mapping [1] (DWM), which groups neighbor lists by size into three categories and uses one thread, one warp, or one block to cooperatively expand one vertex's neighbor list. The strategy achieves good utilization and load balancing within blocks, but can still potentially suffers from intra-block load imbalance

(Merrill et al. provide more details [1]).

- The other alternative is partitioned load-balancing workload mapping [25] (PLB), which ignores any difference in neighbor list size and always chooses to map a fixed amount of vertex or edge workload to one block. When the frontier size is small, it maps a fixed number of vertices to a block (vertex-oriented balanced partitioning). All threads expand all the neighbor lists cooperatively. When the frontier size is large, it maps a fixed number of edges to a block (edge-oriented balanced partitioning). To make all threads cooperatively visit all edges and know to which node's neighbor list each edge belongs requires extra work, either an extra load-balanced search (LBS) or a sorted search [45] is needed after expanding. (Davidson et al. provide more details [25]).

VertexAPI2 uses PT when frontier size is small and uses PLB when frontier size is large. Gunrock dynamically chooses between DWM and PLB according to the graph type (DWM for mesh-like graphs and PLB for scale-free graphs). MapGraph also uses DWM and PLB: PLB for gather and dynamically switching between DWM and PLB for scatter. However, instead of switching between the two according to the graph type, it switches according to the frontier size. Both MapGraph and VertexAPI2 implement PLB using load-balanced search while Gunrock implements PLB using sorted search, both using primitives in Modern GPU [45].

In Figure 4.2 and Figure 4.3, we see significant performance differences between VertexAPI2 and MapGraph, despite both using the same GAS programming model. MapGraph's two-phase method is efficient in expending the small frontiers that commonly appear in long-diameter graphs. VertexAPI2's PT strategy hurts its performance when the frontier size is small, but it spends less time on redundant removal on scale-free graphs. MapGraph behaves similar to Gunrock since similar dynamic strategies are used in both frameworks. However, Gunrock's implementation of PLB saves one pass through the frontier and thus shows superior performance. In general, DWM brings the best performance on small frontiers for both graph types and PLB shows the best performance on large frontiers for scale-free graphs. A successful graph analytic framework must carefully and dynamically consider both graph topology and frontier size to pick the best load-balancing strategy at runtime.



(a) MapGraph (b) VertexAPI2 (c) Gunrock

Figure 4.6: Phase breakdown of graph traversal for three frameworks: Overhead encompasses synchronization and data movement overhead. In MapGraph, the expand phase enumerates neighbors and the contract phase generates next level vertices. VertexAPI2 uses apply to update labels and scatter to activate the next level. In Gunrock, filter is used to generate the next-level frontier while updates can be either in advance or filter, depending on mode

4.3.2 BFS Level Breakdown

Figure 4.6 provides a level breakdown for BFS: in which operations does each framework spend its time? BFS's primary operations are traversing neighbors and updating labels, and its stage breakdown is similar to other graph-traversal-based primitives, like SSSP. We notice that long-diameter road networks and OSMs, which require more bulk synchronized super-steps, have much more synchronization overhead; their execution time is mostly occupied by traversing neighbor lists. Conversely, scale-free and social networks introduce significant redundant edge discovery, thus contract/filter operations dominate their execution time.

4.4 Kernel Execution Patterns

4.4.1 Warp Execution Efficiency

Warp divergence occurs when threads in the same warp take different execution paths. For graph primitives, this is the main contribution of control flow irregularity. Warp execution efficiency (W_{EE}) defines the ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. Table 4.1 shows the average W_{EE} of different graph primitives for three frameworks across four datasets. For BFS primitive, social and scale-free graphs enable a higher average warp execution efficiency across all

Primitive	Framework	road	delaunay	soc	kron
BFS	MapGraph	86.0%	84.1%	84.1%	92.3%
	VertexAPI2	81.6%	73.4%	94.0%	98.4%
	Gunrock	86.6%	83.4%	92.8%	94.1%
SSSP	MapGraph	90.6%	89.1%	94.6%	94.8%
	VertexAPI2	95.4%	95.4%	94.6%	95.9%
	Gunrock	96.9%	96.3%	96.6%	96.6%
CC	MapGraph	95.3%	92.9%	96.7%	97.3%
	VertexAPI2	96.2%	92.3%	95.2%	95.3%
	Gunrock	96.0%	96.8%	98.6%	96.1%
PageRank	MapGraph	96.3%	95.9%	97.1%	98.2%
	VertexAPI2	94.9%	92.4%	97.0%	95.8%
	Gunrock	93.4%	99.5%	99.6%	93.5%

Table 4.1: Average warp execution efficiency (W_{EE})

frameworks due to the highly-optimized load-balanced graph traversal operators used by each framework. However, mesh-like road and Delaunay graphs show lower W_{EE} due to two reasons: 1) underutilization and limited parallelism caused by small frontier sizes and slow frontier size expansion; 2) the use of the per-thread-expand (PT) load-imbalanced neighbor list traversal method. For graph algorithms with dense computation such as PageRank, all three frameworks achieve very high percent W_{EE} because all nodes in the neighbor lists are visited and used for computations. Although CC for GAS is based on BFS, SSSP is traversal-based: all vertices are actively finding minimum neighbors, introducing more parallelism and thus higher W_{EE}.

4.4.2 Synchronization Impacts

Beyond load-balancing, another potential obstacle to performance is the cost of GPU synchronizations, which occur in two places: 1) the implied BSP barriers at the end of each BSP super-step, and 2) implicit global synchronizations between GPU kernel invocations within each super-step. The BSP barrier count is directly proportional to the super-steps required for a primitive to converge, and kernel invocation count corresponds to the number of synchronizations within each super-steps. Each kernel invocation performs four steps: read

		BSP Barrier Count (B_C)				Kernel Invocation Count (K_C)			
	Primitive	MapGraph	VertexAPI2	Gunrock	Hardwired	MapGraph	VertexAPI2	Gunrock	Hardwired
road_usa	BFS	6,263	6,263	6,263	6262	**18,842**	89,045	49,660	11
	SSSP	6,700	6,700	6,700	—	**84,281**	165,967	116,800	—
	CC	6,262	6,262	**13**	10	77,840	178,551	**93**	83
	PageRank	20	20	20	—	1,093	504	**112**	—
delauay-n21	BFS	**564**	565	565	564	2,042	9,229	**1,133**	388
	SSSP	879	879	**871**	—	**10,695**	19,514	14,802	—
	CC	564	564	**7**	7	6,085	14,949	**58**	59
	PageRank	20	20	20	—	889	378	**259**	—
soc-LiveJournal	BFS	**12**	**12**	13	12	144	175	**92**	61
	SSSP	**31**	**31**	32	—	**365**	801	466	—
	CC	12	12	**5**	2	223	292	**46**	22
	PageRank	20	20	20	—	1,094	552	**313**	—
kron_g500-logn21	BFS	**6**	7	**6**	6	96	85	**48**	37
	SSSP	10	10	**9**	—	142	147	**82**	—
	CC	6	6	**5**	4	147	139	**37**	33
	PageRank	20	20	20	—	893	438	**269**	—

Table 4.2: BSP Barrier Count (B_C) and Kernel Count (K_C) of BFS, SSSP, CC, and PageRank for each framework against hardwired implementations [1, 2]: fewer is better. Bold numbers indicate fewest among three programmable frameworks

graph data from global memory, compute, write results to global memory, and synchronize. Many graphs with long tails have substantial synchronization overhead.

Table 4.2 summarizes the BSP Barrier Count (B_C) and Kernel Count (K_C) of BFS, SSSP, CC, and PageRank for each framework against hardwired implementations (BFS: b40c [1]; CC: Soman et al. [2]). All three frameworks share the same BSP model and do not support asynchronous execution, thus each framework has the same number of BSP barriers, except for CC: the huge performance gap between Gunrock and the GAS implementations on CC is primarily from Gunrock's ability to run Greiner's PRAM-based CC algorithm [46], which implements hooking and pointer-jumping using the filter operator. The GAS implementations are instead BFS-based, counting components by graph traversal, and suffer from a large number of synchronizations and slow information propagation, especially for long-tail graphs.

Both B_C and K_C show strong positive correlations with achieved performance. The overall correlation is shown in Table 4.3: for traversal-based primitives where all three frameworks

	BFS	SSSP	CC	PageRank	Overall
Correlation B_C	0.759	0.721	0.878	—	0.624
Correlation K_C	0.868	0.577	0.615	0.308	0.575

Table 4.3: Correlation of B_C and K_C with performance (measured as throughput). PageRank's B_C data is not applicable because of its fixed super-step count

follow similar approaches, fewer synchronizations (implying kernels that do more work) yield superior performance. Expert programmers fuse kernels [1, 47] together to reduces synchronization and increase producer-consumer locality by reducing reads and writes to global memory.

However, reducing kernel invocations is a non-trivial task for programmable frameworks because the building blocks of programmable frameworks—operators—are typically kernel invocations. Generally, high-level programming models trade off increased kernel invocation overhead (compared to hardwired implementations) for flexibility and more diverse expressiveness. That being said, reducing kernel invocations K_C is a worthwhile target for any framework implementation, and Gunrock's ability to fuse compute operations into advance or filter kernels appears to directly translate into both fewer kernel invocations and thus overall better performance.

4.5 Memory Behavior Impacts

As graph primitives are often memory-bound due to a lack of locality, factors such as data movement, memory access patterns, and total memory usage all have obvious impacts on achieved performance.

4.5.1 Atomic Operations

Atomic instructions are generally considered expensive on GPUs [48], although their cost has decreased with more recent GPU micro-architectures. Unfortunately, atomic operations are a key ingredient of operations on irregular graph data structures, due to the large amount of concurrent discovery particularly characteristic of scale-free graphs. Table 4.4 summarizes each framework's number of global atomic operations. BFS's/SSSP's atomics are found

Primitive	Dataset	MapGraph	VertexAPI2	Gunrock
BFS	roadNet-CA	55	320	56
	kron_g500-logn17	8,091	1,723	2,862
SSSP	roadNet-CA	4,516	60,413	3,956
	kron_g500-logn17	454,973	97,394	2,211
CC	roadNet-CA	288,309	112,270	15,657
	kron_g500-logn17	2,569,069	162,983	1,082
PageRank	roadNet-CA	0	0	7,680
	kron_g500-logn17	0	0	212

Table 4.4: Atomic operations in each framework: fewer is better

in MapGraph's expand/contract, VertexAPI2's activate, and Gunrock's filter. MapGraph and Gunrock show similar atomic behavior because they both visit neighbor lists and do push-updates, then contract/filter out redundant vertices; for both, the work of contract/filter is substantial. VertexAPI2 uses more atomic operations on road networks because its simple per-thread expanding atomic activates are frequently used; conversely, its fewer atomic operations in scale-free graphs are due to VertexAPI2's use of a Boolean flag to indicate vertex status in the new frontier-generating phase (activate).

Gunrock results in this thesis incorporate *idempotent* optimization, applicable primarily to BFS, that allows multiple insertions of the same vertex in the frontier without impacting correctness. It reduces the atomic operation count from 20,034 to 2,862 for BFS on `kron_g500-logn17`. On this dataset, the idempotent optimization improves the traversal rate in MTEPS from 891.7 MiEdges/s to 3291 MiEdges/s (a 3.69X speedup). Without the idempotent operation, using atomic operations to check whether or not the vertices in next level have already been claimed as someone else's child is extremely expensive. In GAS model, this optimization is not applicable because GAS frameworks cannot guarantee the idempotent of arbitrary user-defined apply functions [32]. Turning to PageRank, we note that PageRank's runtime is dominated by vertex-centric updates and ideally suited to the GAS abstraction; neither VertexAPI2 nor MapGraph requires any atomic operations in their implementations, and partially as a result, both frameworks deliver excellent performance.

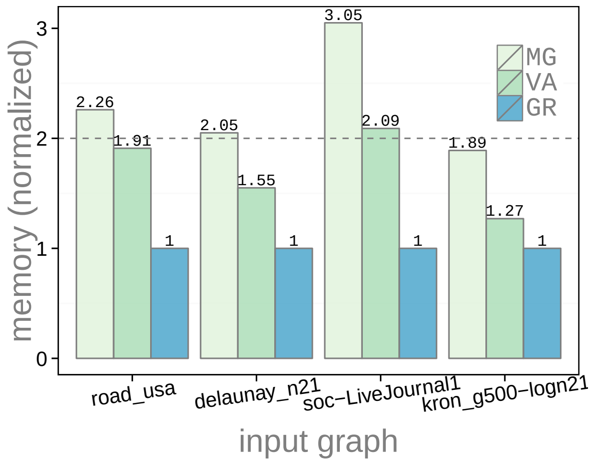
Framework	MapGraph		VertexAPI2		Gunrock	
dataset	load	store	load	store	load	store
delaunay	422k	204k	23.5k	9.78k	1.48k	4.70k
kron	18.5M	981k	840k	8.73M	1.89M	736k

Table 4.5: Achieved global load and store transactions on BFS: `delanuay_n17` contains 131k vertices and 393k edges, and `Kron_g500-logn17` contains 131k vertices and 10.1M edges

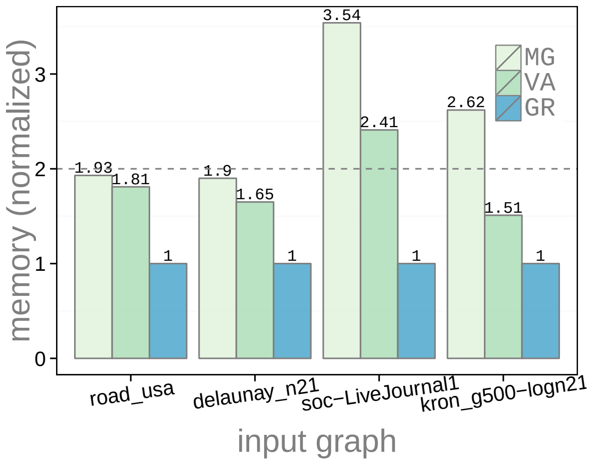
4.5.2 Data Movement

Another metric directly proportional to kernel invocations of data-intensive graph primitives is the data movement across GPU device kernels. Let the number of vertices in the graph be $|V|$ and the number of edges $|E|$, and consider BFS in both programming models as an example: GAS model requires $5|E| + 6\ |V|$ data transfers of which $4|E| + 3|V|$ are coalesced. In the data-centric model, both of Gunrock's load-balancing strategies require $5|E| + 4|V|$ global data transfers, of which $3|E| + |V|$ are coalesced.

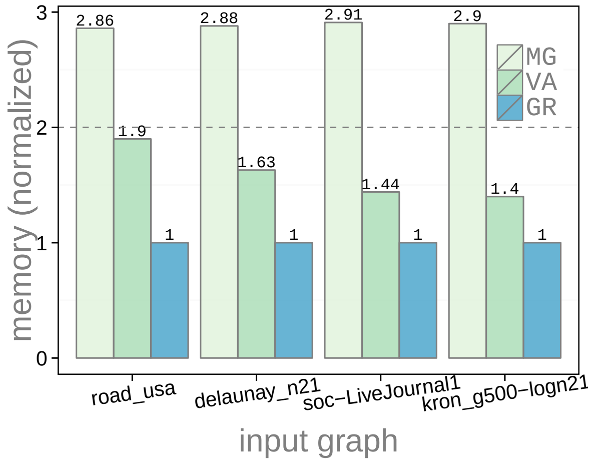
Table 4.5 summarizes the actual global load and global store counts for the three frameworks running BFS on two types of real-world graphs. We notice that for `delaunay_n17`, MapGraph requires much more data communication than Gunrock and VertexAPI2. Potential reasons for this behavior are: 1) the cost of maintaining an edge frontier between MapGraph's expand and contract phase, and 2) the requirement for two-level loads from global memory to registers: first loading frontier data into tile arrays and then loading from those arrays to do data computation. In contrast, Gunrock's PLB only has one level of such loads. VertexAPI2, on the other hand, uses flags to indicate the status of vertices without generating edge frontiers, which results in fewer data transfers. On `kron_g500-logn17`, MapGraph and Gunrock both have many more reads than writes due to heavy concurrent discovery. In general, performance is consistent with the number of memory transactions. The framework which has the least amount of memory transactions usually have the best performance, such as Gunrock on road networks and VertexAPI2 on kron.



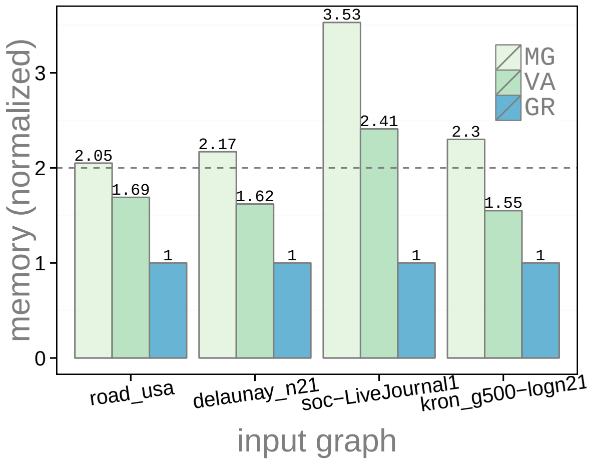
(a) Memory Consumption: BFS



(b) Memory Consumption: SSSP



(c) Memory Consumption: CC



(d) Memory Consumption: PageRank

Figure 4.7: Memory consumption across datasets (normalized to Gunrock = 1)

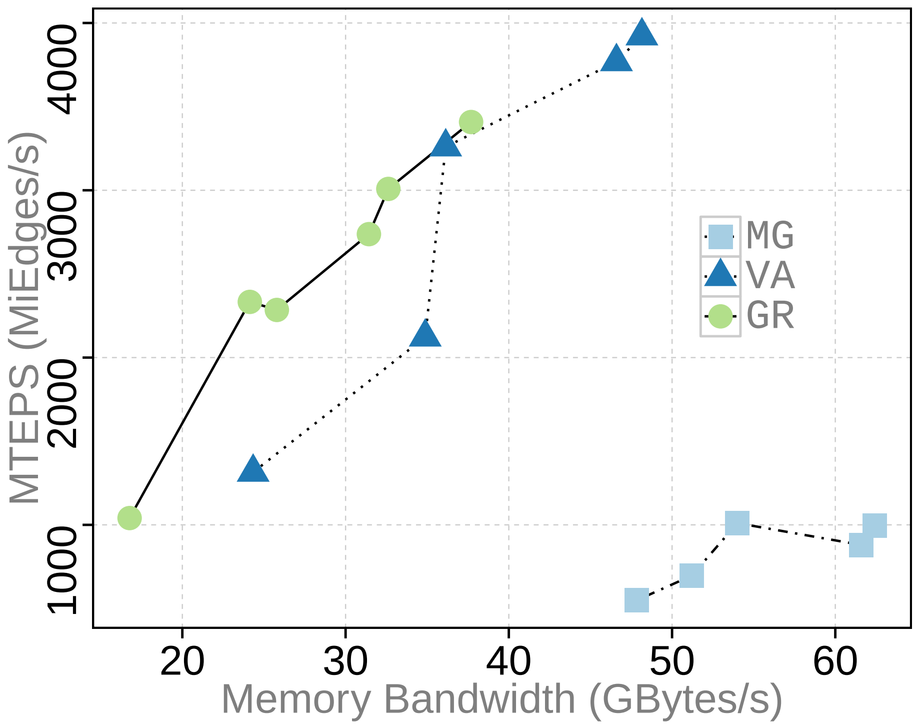


Figure 4.8: Impact of global memory bandwidth on performance

4.5.3 Memory Consumption and Bandwidth

All frameworks use Compressed Sparse Row (CSR) format or CSR-based format (Section 2.1) to store graphs in memory. The design choice of different CSR-based graph formats of each framework and how to efficiently access them dramatically affects performance. This section will focus on the characterization of memory usage and bandwidth.

BFS is memory-bandwidth-limited for large graphs and therefore bandwidth has to be handled with care. Without losing generality, Figure 4.8 illustrates the traversal throughput, MTEPS, as a function of memory bandwidth running BFS on five kron datasets (`kron_g500-logn17` ∼ `kron_g500-logn21`). All three frameworks follow the same patterns: higher achieved memory bandwidth leads to higher throughput. VertexAPI2 and Gunrock use memory bandwidth more efficiently. However, no framework approaches our GPU's theoretical maximum memory bandwidth of 288 GB/sec, which implies that implementations based on current CSR-based graph representations utilize a substantial number of scattered (uncoalesced) reads and writes.

The frameworks in this study abstract graph primitives with iterative advance + filter steps or gather + apply + scatter steps. In the data-centric model, the advance traversal uses CSR to expand the neighbors of the current vertex frontier. In GAS, the gather phase requires a CSC representation of a graph to gather its neighbors (pull), and GAS's scatter phase requires CSR (push) to complete push-style updates and/or activations. Thus the GAS implementations require storing the graph topology in both CSR and CSC formats, when we have directed input. The usage of CSC doubles the memory consumption of GAS implementations compared to Gunrock for directed inputs. On the other hand, Gunrock integrates optimizations that consume memory beyond only CSR, such as edge list expansion in CC and direction-optimized graph traversal ("pull") on BFS, which requires a CSC format. Gunrock's pull-enabled BFS implementation requires 3.52 GB on `kron_g500-logn21` dataset (1.26X compared with non-pull-enabled implementation). This optimization improves the achieved traversal rate from 3476.6 MiEdges/s to 8384.5 MiEdges/s. Here, Gunrock demonstrates a trade-off between memory consumption and performance.

In Figure 4.7, we investigate the global memory consumption. Scale-free graphs often consume additional memory compared to long-diameter graphs due to the cost of maintaining the fast-expanding and extremely large edge frontiers. In the GAS implementations, MapGraph uses a two-step procedure to build CSR and CSC formats in memory, which result in an overall factor of 1.88X $\sim$ 3.54X above Gunrock. VertexAPI2's implementation eliminates the scatter phase, which result in less but still not optimal memory consumption (1.27 $\sim$ 2.4 compared to Gunrock). The memory footprint for GAS abstraction and implementation is clearly not optimal. Running BFS on `kron_g500-logn21` requires 2.81 GB for Gunrock, compared to 7.36 GB for MapGraph. Ultimately the large memory consumption of the GAS implementations limits the size of graphs that the GAS implementations can fit into the GPU's limited memory (12 GB on the K40c). In real-word graph analytics problems, we should not only consider the graph as simple collection of vertices and edges, the possible rich and diverse information can be associated with each vertex or edge will result in more complexity in terms of both storage and processing.

Chapter 5

Discussion and Future Work

High-level GPU programmable interfaces are crucial for programmers to quickly build and evaluate complex workloads using graph primitives. However, graphs are particularly challenging: the intersection of varying graph topologies with different primitives yield complex and even opposing optimization strategies. In this work we have learned that the Gather-Apply-Scatter (GAS) abstraction can eliminate expensive atomic operations using synchronous pull-based implementation to reduce synchronization overhead for primitives dominated by vertex-centric updates; however, it suffers from slow information propagation and high memory consumption. Gunrock's data-centric abstraction enables more powerful and flexible operators and lower memory consumption. In practice, Gunrock's implementation allows integrating more work in each kernel, thus requiring fewer kernel invocations, synchronizations, data movements, and resulting in higher achieved throughput. For any of the frameworks we studied, and for future frameworks, the design choice of operators and the ability to implement efficient algorithms are key to achieving best-of-class performance. From an architecture perspective, better hardware and programming-system support for load-balancing irregular parallelism would be a worthwhile investment for better support of graph analytics.

5.1 Graph Representation and Mutability

All three frameworks utilize the CSR graph format (Section 2.1), and they do not provide any mutability: one cannot easily add or remove vertices and/or edges from a CSR graph. The benefit of CSR are fast traversal and memory efficient, however, changing topology is an extremely expensive operation during graph computation. Lack of support for mutable graph structure limits the expressiveness of graph analytics; how to extend GPU frameworks to support mutable topology is future work. For instance, the implementation of forming a *"super-vertex"*, group a subset of vertices according to some user-defined criteria into one super-vertex for next-level computations, commonly used in more complex primitive such as in Borůvka's Minimum Spanning Tree (MST) [49] and community detection [50] algorithms requires rebuilding the entire in-memory CSR arrays for each super-step. Generalized approaches for supporting dynamic graph topology changes during the computation on GPUs can improve both programmability and efficiency. The static graph also limits the possible solution to incrementally computations, with the slight modification of input graph such as changing one edge or removing some vertices/edges. Current frameworks would recompute using the new graph rather than incrementally update based on the modifications. Building data structures for efficient storage, access, and updates of graph information is still an unsolved challenge. Alternate graph formats might allow superior memory performance, particularly with regard to data coalescing.

5.2 Powerful Operator Design

Graph analytics requires more powerful yet efficient operators that can manipulate graph computations in varieties of ways. Common operators should be extended beyond simple graph traversal and apply/compute. Other operations that appear many times in more advanced primitives should be supported such as *"pointer-jumping"* (group a subset of vertices into a star-like graph with one representative) used in CC and MST; forming a super-vertex used in MST, community detection, and many others. How to keep improving the generality of the programmable interface meaning high performance is worth exploring.

5.3 Asynchronous Execution

None of the programmable frameworks show impressive results on low-degree long-tail graphs. More broadly, the common challenges to the frameworks we studied include synchronization cost and limited parallelism. These may be a limitation of the BSP model and concentrate on level-synchronous algorithms common to all three frameworks. The consistent synchronous approach can bring significant overheads and penalties due to the requirement of locking or atomic operations. Using an asynchronous execution framework may be an interesting direction for future work. Recent effort Frog [51] explores the possible ways to enable asynchronous graph processing on the GPU with a graph coloring model, updating all the vertices with the same color in parallel (one color per kernel execution). They show improvements by enabling asynchronous executions at the price of non-trivial preprocessing (partition the graph). Better support for asynchronous operations is an important future direction.

5.4 Other Challenges

Another challenge is automatic kernel fusion, which can potentially reduce synchronization cost [1, 47], but current GPU programming frameworks do not perform this (difficult) optimization automatically. Memory performance is a crucial aspect of any GPU graph primitive. Finally, scalability is an important aspect that has not been covered in this performance characterization works, however, it is vital for multi-GPU extension as real-world graph size increases.

References

[1] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, Feb. 2012, pp. 117–128, doi: 10.1145/2145816.2145832. (document), 1.2, 2.2, 3.1.2, 4.3.1, 4.2, 4.4.2, 5.4

[2] J. Soman, K. 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*, ser. IPDPSW 2010, Apr. 2010, pp. 1–8, doi: 10.1109/IPDPSW.2010.5470817. (document), 1.2, 2.2, 3.1.2, 4.2, 4.4.2

[3] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, May 2013, pp. 825–836, doi: 10.1109/IPDPS.2013.72, ISBN: 978-0-7695-4971-2. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.72 1.1

[4] A. McLaughlin and D. 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*, ser. SC14, Nov. 2014, pp. 572–583, doi: 10.1109/SC.2014.52, ISBN: 978-1-4799-5500-8. 1.1, 1.2, 3.1.2

[5] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14, May 2014, pp. 889–901, doi: 10.1109/IPDPS.2014.96. 1.1

[6] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri, "FAST-PPR: Scaling personalized PageRank estimation for large graphs," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, Aug. 2014, pp. 1436–1445, doi: 10.1145/2623330.2623745, ISBN: 978-1-4503-2956-9. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623745 1.1

[7] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC13, Nov. 2013, pp. 92:1–92:11, doi: 10.1145/2503210.2503246, ISBN: 978-1-4503-2378-9. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503246 1.1

[8] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46. 1.1

[9] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Nov. 2013, pp. 456–471, doi: 10.1145/2517349.2522739. 1.1

[10] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488, doi: 10.1145/2517349.2522740, ISBN: 978-1-4503-2388-8. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740 1.1

[11] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 77–85, doi: 10.1145/2487575.2487581, ISBN: 978-1-4503-2174-7. [Online]. Available: http://doi.acm.org/10.1145/2487575.2487581 1.1

[12] J. Shun and G. 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*, ser. PPoPP '13, Feb. 2013, pp. 135–146, doi: 10.1145/2442516.2442530. 1.1

[13] J. Shun, L. Dhulipala, and G. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *Proceedings of the IEEE Data Compression Conference (DCC)*, 2015. 1.1

[14] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 183–193, doi: 10.1145/2688500.2688507, ISBN: 978-1-4503-3205-7. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688507 1.1

[15] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Dec. 2001. 1.1

[16] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," in *Parallel Object-Oriented Scientific Computing (POOSC)*, Jul. 2005. 1.1

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, Jun. 2010, pp. 135–146, doi: 10.1145/1807167.1807184. 1.1

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “GraphLab: A new parallel framework for machine learning,” in *Proceedings of the Twenty-Sixth Annual Conference on Uncertainty in Artificial Intelligence*, ser. UAI-10, Jul. 2010, pp. 340–349. 1.1, 2.3.1

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012. 1.1

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed graph-parallel computation on natural graphs,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’12. USENIX Association, Oct. 2012, pp. 17–30. 1.1, 2.3.1

[21] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “GraphX: Graph processing in a distributed dataflow framework,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613, ISBN: 978-1-931971-16-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685096 1.1

[22] S. Salihoglu and J. Widom, “HelP: High-level primitives for large-scale graph processing,” in *Proceedings of the Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES ’14, Jun. 2014, pp. 3:1–3:6, doi: 10.1145/2621934.2621938. 1.1

[23] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, “GPUs and the future of parallel computing,” *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011, doi: 10.1109/MM.2011.89. 1.2

[24] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek, “Betweenness centrality on GPUs and heterogeneous architectures,” in *GPGPU-6: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, Mar. 2013, doi: 10.1145/2458523.2458531. 1.2, 3.1.2

[25] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-efficient parallel GPU methods for single source shortest paths,” in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2014, May 2014, pp. 349–359, doi: 10.1109/IPDPS.2014.45. [Online]. Available: http://escholarship.org/uc/item/8qr166v2 1.2, 3.1.2, 4.3.1

[26] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in *IEEE International Symposium on Workload Characterization*, ser. IISWC 2012, Nov. 2012, pp. 141–151, doi: 10.1109/IISWC.2012.6402918. 1.2, 1.3, 3.1.4

[27] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, “Pannotia: Understanding irregular GPGPU graph applications,” in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195. 1.3, 3.1.2, 4.3

[28] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *IEEE International Symposium on Workload Characterization*, ser. IISWC-2014, Oct. 2014, pp. 140–149, doi: 10.1109/IISWC.2014.6983053. 1.3, 3.1.2, 4.3

[29] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *IEEE International Symposium on Workload Characterization*, ser. IISWC-2014, Oct. 2014, pp. 130–139, doi: 10.1109/IISWC.2014.6983052. 1.3

[30] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, no. 1501.05387v2, Mar. 2015. 1.2, 2.3, 4.3

[31] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014, doi: 10.1109/TPDS.2013.111. 1.2

[32] E. Elsen and V. 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. 1.2, 2.3, 4.3, 4.5.1

[33] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A high level API for fast development of high performance graph analytics on GPUs," in *Proceedings of the Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES '14, Jun. 2014, pp. 2:1–2:6, doi: 10.1145/2621934.2621936. 1.2, 2.3, 4.3

[34] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, Jun. 2014, pp. 239–252, doi: 10.1145/2600212.2600227, ISBN: 978-1-4503-2749-7. 1.2

[35] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," *CoRR*, vol. abs/1504.04804, no. 1504.04804v1, Apr. 2015. 1.2

[36] E. S.-N. A. Gharaibeh, L. B. Costa, and M. Ripeanu, "Totem: Accelerating graph processing on hybrid CPU+GPU systems," in *NVIDIA GPU Technology Conference 2013*, Mar. 2013. 1.2

[37] F. Yang and A. A. Chien, "Understanding graph computation behavior to enable robust benchmarking," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15, 2015, pp. 173–178, doi: 10.1145/2749246.2749257, ISBN: 978-1-4503-3550-8. 1.3

[38] K. A. Scott Beamer and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *IEEE International Symposium on Workload Characterization*, ser. IISWC-2015, Oct. 2015. 1.3

[39] J. Shun, "An evaluation of parallel eccentricity estimation algorithms on undirected real-world graphs," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 1095–1104, doi: 10.1145/2783258.2783333, ISBN: 978-1-4503-3664-2. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783333 2.2

[40] S. Che, "GasCL: A vertex-centric graph model for GPUs," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014. 2.3

[41] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. 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*, ser. PPoPP 2015, Feb. 2015, pp. 271–272, doi: 10.1145/2688500.2688542, ISBN: 978-1-4503-3205-7. 3.1.3

[42] J. H. Kim and V. H. Vu, "Generating random regular graphs," in *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '03. New York, NY, USA: ACM, 2003, pp. 213–222, doi: 10.1145/780542.780576, ISBN: 1-58113-674-9. [Online]. Available: http://doi.acm.org/10.1145/780542.780576 3.2, 3.2, 4.2.2

[43] D. A. Bader and K. Madduri, "GTgraph: A suite of synthetic graph generators," 2006, https://github.com/dhruvbird/GTgraph. 3.2, 3.2, 4.2.2

[44] T. A. Davis, "The University of Florida sparse matrix collection," *NA Digest*, vol. 92, no. 42, 16 Oct. 1994, http://www.cise.ufl.edu/research/sparse/matrices. 3.2

[45] S. Baxter, "Modern GPU," https://nvlabs.github.io/moderngpu/, 2013. 4.3.1

[46] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '94, Jun. 1994, pp. 16–25, doi: 10.1145/181014.181021. 4.4.2

[47] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan, "On optimizing machine learning workloads via kernel fusion," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 173–182, doi: 10.1145/2688500.2688521, ISBN: 978-1-4503-3205-7. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688521 4.4.2, 5.4

[48] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 96–107, doi: 10.1145/2458523.2458533, ISBN: 978-1-4503-2017-7. [Online]. Available: http://doi.acm.org/10.1145/2458523.2458533 4.5.1

[49] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” in *Proceedings of High Performance Graphics*, ser. HPG ’09, Aug. 2009, pp. 167–171, doi: 10.1145/1572769.1572796. 5.1

[50] S. Fortunato, “Community detection in graphs,” *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010. [Online]. Available: http://arxiv.org/pdf/0906.0612.pdf 5.1

[51] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. 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*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 271–272, doi: 10.1145/2688500.2688542, ISBN: 978-1-4503-3205-7. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688542 5.3