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

Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens
Electrical and Computer Engineering, University of California, Davis, CA 95616
{yudwu, yzhwang, ychpan, ctcyang, jowens}@ucdavis.edu

Abstract— **We identify several factors that are critical to high-performance GPU graph analytics: efficient 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 GPU graph analytic frameworks, Gunrock, MapGraph, and VertexAPI2. 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 the GPU.**

I. Introduction

Large-scale graph structures with millions or billions of vertices and edges are rapidly created by modern applications. Graph analytics 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, and biological networks to targeted advertising. The demands of diverse graph applications have led to numerous efforts on speeding up graph primitives via parallelization architectures. A number of shared-memory or distributed CPU graph frameworks focusing on efficiency have emerged in recent years [1–5]. Other work [6] 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 or other frameworks. These graph analytic frameworks allow programmers to concentrate on expressing primitives, because the framework takes care of automatically scaling the computations on parallel architectures.

Graphics processing units (GPUs) are power-efficient and high-memory-bandwidth processors that can exploit parallelism in computationally demanding applications. GPUs have proved to be extremely effective at accelerating operations on traditional vector- or matrix-based data structures, which exhibit massive parallelism, regular memory access patterns, few synchronizations, and a straightforward mapping to parallel hardware. Unfortunately, mapping irregular graph primitives to parallel hardware is non-trivial due to the data-dependent control flow and unpredictable memory access patterns [7]. Recent low-level hardwired implementations [8–11] have demonstrated the strong computational power of modern GPUs in bandwidth-hungry graph analytics. For more general real-world graph analytics, developers need high-level programmable frameworks to implement various types of complex graph applications on the GPU without sacrificing much performance. Programming models are key design choices that impact both expressivity and performance. Most GPU+graph programming models today mirror CPU programming models: for instance, Medusa [12] uses Pregel's message passing model, and VertexAPI2 [13], MapGraph [14], and CuSha [15] use and modify PowerGraph's Gather-Apply-Scatter (GAS) model. A more recent framework, Gunrock [7], uses a GPU-specific data-centric model focused on operations of a subset of vertices and/or edges.

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. A thorough investigation of graph framework performance on a single GPU 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. Our contributions are as follows:

1) 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.
2) 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.
3) 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.

The rest of this paper is organized as follows: Section II reviews related work for graph application characterizations. Section III first provides the necessary background on graph analytics using high-level frameworks, then reviews several common graph primitives used in this work as case studies. In Section IV, we discuss the two existing graph programming models and introduce three implementations on the GPU. Section V defines the methodology and performance metrics used to investigate the programming models. Detailed empirical studies are provided in Section VI. Section VII discusses potential ways to further improve graph analytic abstractions.

II. Related Work

Developing and evaluating graph primitives on GPU is a hot recent topic. Xu et al. [16] 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 [17] 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 SIMT lane utilization. O'Neil et al. [18] 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. [19] 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 [20] studied different ensembles of parallel graph computations, and concluded that graph computation behaviors form an extremely broad space.

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.

III. Preliminaries & Background

A. Selection of Graph Topologies

The performance of graph primitives is highly data-dependent, as we describe in more detail in Sec. VI-A. We explore how topology impacts the overall performance using the following metrics. The *eccentricity* $\epsilon(v)$ is the greatest geodesic distance between a vertex v and any other vertex in the graph. The *radius* of a graph is the minimum graph eccentricity $r = \min\ \epsilon(v)$ and in contrast, the *diameter* of a graph is the length of the maximum over shortest paths between any pair of vertices $d = \max\ \epsilon(v)$. The magnitude of *algebraic connectivity* reflects the well-connectedness of the overall graph. For traversal-based graph primitives, traversal *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; an unbalanced degree distribution can significantly impact load balancing during the 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 Section VI, we choose diverse datasets that encompass both categories, and also generate several synthesized graphs whose eccentricity and diameter values span from very small to very large.

B. Selection of 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, 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/all vertices in a *dense-computation-based* primitive are active in every stage of the computation.

Breath-First Search (BFS) is a common building block for more sophisticated graph primitives, and is representative of a class of irregular and data-dependent parallel computations [9]. Other traversal-based primitives include Betweenness Centrality (BC), a quantitative measure to capture the effect of important vertices in a graph, and Single-Source Shortest Path (SSSP), which calculates the shortest distance between the source $v_{\text{source}} \in V$ and all vertices in a weighted graph $G = (V, E, w)$. These three primitives are generally considered traversal-based. Depending on its implementation, Connected Component (CC) can be in either category: 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 $V \backslash C$. Link analysis and ranking primitives such as PageRank calculate the relative importance of vertices and are dominated by vertex-centric updates; PageRank is a dense-computation-based primitive.

In this work, we use four common primitives—BFS, SSSP, CC and PageRank—as case studies to benchmark the performance of 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 graph analytics.

IV. Programming Models Overview

Many graph primitives can be expressed as several inherently parallel computation stages interspersed with synchronizations. Existing programmable frameworks on the GPU [13, 14, 7, 21] all employ a Bulk-Synchronous Parallel (BSP) programming model in which users express a graph problem a series of consecutive "*super-steps*", separated by global barriers, where each super-step exhibits ample data parallelism and can be run efficiently on a data-parallel GPU. These super-steps may include per-vertex and/or per-edge computation that run on all or a subset of vertices or edges in the graph, for instance, reading and/or updating each vertex's/edge's own data or that of its neighbors. A super-step may instead traverse the graph and choose a new subset of active vertices or edges (a "*frontier*") during or after graph computations. Thus the data parallelism within a super-step is typically data parallelism over the vertices or edges in the frontier. The programs themselves are generally iterative, convergent processes that repeatedly run super-steps that update vertex/edge values 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 generate a new frontier for the next super-step. In this work, we study VertexAPI2 [13], an implementation strictly following the Gather-Apply-Scatter (GAS) model; MapGraph [14], a modified version of the GAS model; and Gunrock [7], a data-centric abstraction that focuses on manipulations of the frontier. All three frameworks run bulk-synchronous super-steps on a frontier of active vertices/edges until convergence. In the rest of the paper, we use VA to denote VertexAPI2, MG for MapGraph, and GR for Gunrock.

A. Gather-Apply-Scatter Model

The Gather-Apply-Scatter (GAS) approach was originally developed for distributed environments [2, 1]. 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. 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 VA and MG broadly follow the GAS model, but with important differences. VA 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, VA writes predicate values to a v-length flag, then invokes an *activate* phase to scan and compact vertices associated with true flags to create frontiers for the next iteration. MG 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 MG support in-edges, out-edges, or both.

B. Data-Centric Model

Rather than focusing on expressing sequential steps of computation on vertices, GR's abstraction focuses on manipulations of the frontier of vertices or edges that are actively participating in the computation. GR 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. GR's advance and filter dynamically choose optimization strategies during runtime depending on graph topology. In order to reduce the number of kernel calls and enhance producer-consumer locality, GR's computation steps are expressed as device functions called functors that can be fused into advance and filter kernels at compile time. Fig. 1 shows how we can express BFS using operators in three different frameworks.

V. Critical Aspects 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.

Efficient 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. We evaluate both in this work. 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 IV include gather, apply, expand, contract, activate, advance, and filter.

Workload distribution and load balancing are crucial issues for performance; previous work has observed that these operations are dependent on graph structure [17, 16]. Hardwired graph primitive implementations 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; GR'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.

Synchronization and data movement are limiting factors under widely-adopted BSP models. While the BSP model of computation is a good fit for the 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 [22]. 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 kernels. How do the high-level programming models we consider here, and the implementations of their operations, impact performance?

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 [19]. 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.

Listing 1: BFS using MG's API

```
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: BFS using VA's API

```
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 3: BFS using GR's API

```
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
```

Fig. 1: Code snapshots for three different GPU graph analytic frameworks.

VI. Methodology & Evaluation

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 of graph traversal and information propagation. Then to better understand the performance, we focus on the following details of the abstraction and implementations: load balance and work distribution, synchronization, and memory usage and behavior.

We empirically evaluate the effectiveness of primitives (Section III-B), expressed in the GAS implementations of VA v2 and MG v0.3.3 and the data-centric implementation of GR[1] (Section IV), on our benchmark suite. All experiments ran on a Linux workstation with 2 × 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, and a NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVCC v6.5.12 and C/C++ code was compiled using GCC v4.8.1 with the -O3 optimization level. In this work, we aim to focus on an abstraction-level understanding of the frameworks centered on their GPU implementations; thus all results ignore transfer time (disk-to-memory and host-to-device). We use the NVIDIA Visual Profiler to collect some of our characterization results.

Table I summarizes our benchmark suite. Road and Open Street Maps (OSM) are two types of real-world road networks with most (at least 87.1%) vertices having an directed out-degree below 4. Delaunay datasets are Delaunay triangulations of random points in the plane that also have extremely small out-degrees. Social networks commonly have scale-free vertex degree distributions and small diameters (thus small depths). Kronecker datasets are similar to social networks with the majority of the vertices belonging to only several levels of BFS. Overall, these datasets cover a wide range of topologies to help us characterize and understand the performance and effectiveness of different graph analytic frameworks. Most of our datasets are from the University of Florida Sparse Matrix Collection [23]; we also use synthetic graph generators [24, 25]. To provide weights for the SSSP, we associate a random integer weight in the range of [1, 128) to each edge.

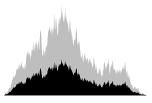
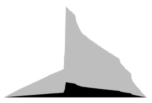
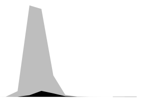
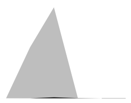
Dataset	Vertices	Edges	$\overline{Degree}$	$\overline{Depth}$	BFS Frontier Size
roadNet-CA	1.97M	5.53M	2.81	657	
asia_osm	12.0M	25.4M	2.13	36,077	
road_central	14.1M	33.9M	2.41	4,208	
road_usa	23.9M	57.7M	2.41	6,155	
europe_osm	51.0M	108M	2.12	19,338	
delaunay_n17	131k	393k	6.00	155	
delaunay_n18	262k	786k	6.00	214	
delaunay_n19	524k	1.57M	6.00	295	
delaunay_n20	1.05M	3.15M	6.00	413	
delaunay_n21	2.10M	6.29M	6.00	571	
amazon-2008	735k	7.05M	9.58	21	
hollywood-2009	1.14M	113M	98.9	8	
tweets	1.85M	5.75M	3.12	16	
soc-orkut	3.00M	213M	71.0	8	
soc-LiveJournal1	4.85M	85.7M	17.7	14	
kron_g500-logn17	131k	10.1M	78.0	3	
kron_g500-logn18	262k	21.0M	80.7	3	
kron_g500-logn19	524k	43.2M	83.1	4	
kron_g500-logn20	1.05M	88.6M	85.1	5	
kron_g500-logn21	2.10M	181M	86.8	5	

TABLE I: 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. The last column depicts the size of vertex frontiers (black) and edge frontiers (gray) as a function of BFS level on road_usa, delaunay_n21, soc-LiveJournal1, and kron_g500-logn21. 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. 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.

A. Overall Performance

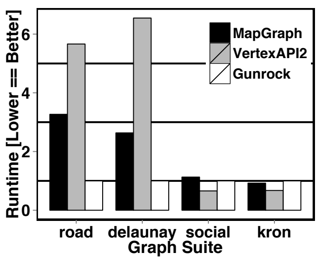
Fig. 2 contains the normalized runtime for four graph primitives across all datasets on three frameworks. We observe that: 1) The runtime ratios of a primitive evaluated on the three frameworks can heavily differ simply because of the topology of the input graph; 2) All frameworks have much higher traversal rates on scale-free graphs than 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 observed differences in overall performance.

B. Dataset Impact

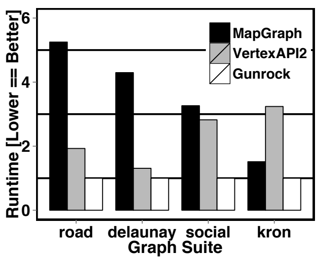
Ample parallelism is critical for GPU computing. To make full use of the GPU, programmers must supply sufficient work to the GPU to keep its many functional units occupied with enough active threads to hide memory latency. The NVIDIA K40c we used here has 15 multiprocessors, each of which can support 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

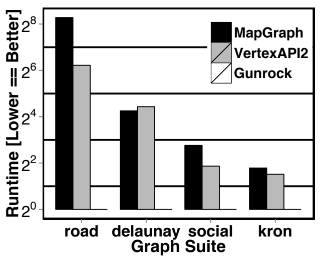
[1]Gunrock's results use git commit 3f1a98a dated May 8, 2015.



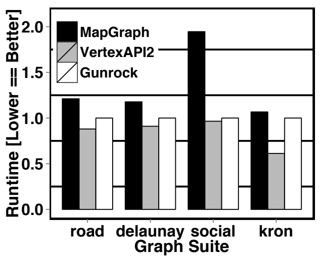
(a) Breadth-First Search



(b) Single-Source Shortest Path

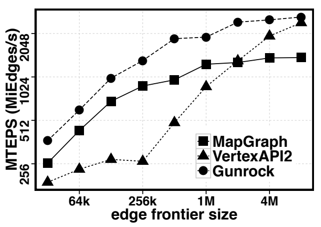


(c) Connected Component

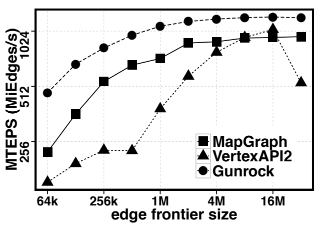


(d) PageRank

Fig. 2: Runtime of BFS, SSSP, CC, and PageRank, normalized to GR's runtime. For a fair comparison, all GR BFS results disable the direction-optimizing (pull updates) optimization.

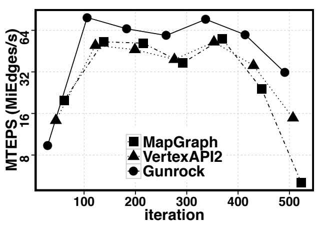


(a) Breadth-First Search

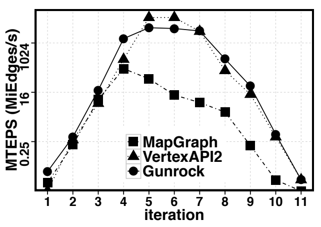


(b) Single-Source Shortest Path

Fig. 3: Traversed edges per second vs. edge frontier size, measured on synthetic trees with a perfectly balanced workload and no redundant work (no concurrent child discovery).



(a) roadNet-CA



(b) hollywood-2009

Fig. 4: Traversed edges per second for each iteration of two types of real-world graphs. Note that roadNet-CA is sampled.

traversal: two traversal-based primitives, BFS and SSSP, on a 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. Fig. 3 (a) illustrates the throughput in millions of traversed edges per second (MTEPS) as a function of the edge frontier size at each BFS level. Generally, throughput improves 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, high-diameter graphs like road networks will be unlikely to achieve maximum throughput on BSP-based GPU graph frameworks.

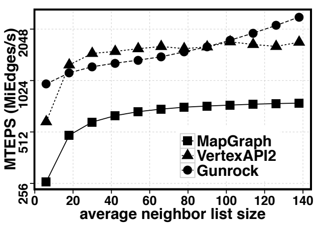
VA's saturation point exceeds the largest synthetic frontier size we used. We attribute such scalability and performance boost for VA to their switching of parallelism strategy beyond a certain frontier size. These behavioral patterns suggest that ample workload is critical for high throughput, and thus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

Let's now turn to two real-world graphs: a high-diameter road network, roadNet-CA, and a scale-free social graph, hollywood-2009. We show MTEPS for these graphs in Fig. 4. 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 ($\approx 2^{26}$), which is substantially larger than the saturation frontier size we observed. The performance patterns are consistent with the previous key findings; VA achieves peak performance for 2 iterations of the largest frontiers, which explains its behavior on social and kron.

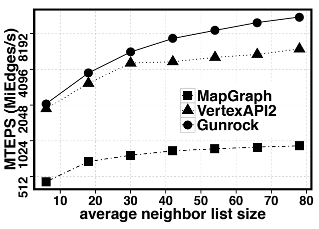
BFS and SSSP are traversal-based and thus only have a subset of vertices or edges active in a frontier at any given time. We see that when that fraction is small, we do not achieve peak performance. In contrast, a dense-computation-based primitive like PageRank has all vertices active in each iteration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

Another important characteristic of graphs is the average degree, which we can use to quantify per-vertex workload. Fig. 5 shows the performance impact of the average degree of BFS with a set of synthetic regular graphs [24] and scale-free graphs (R-MAT [25]), all with the same number of vertices and only differing in average degree. All frameworks demonstrate better performance with increasing degree; more work per vertex leads to more throughput. Smaller average degree graphs (e.g., road networks and OSMs) are limited by available parallelism. Larger average degree (over 20$\sim$30) graphs demonstrate notably higher MTEPS for all frameworks. In practice, this larger degree is necessary to provide enough work to keep the GPU fully occupied.

Finally, scale-free vertex distributions and their skewness also impact performance. We can broaden our analysis by varying parameters of the R-MAT generator to create not only

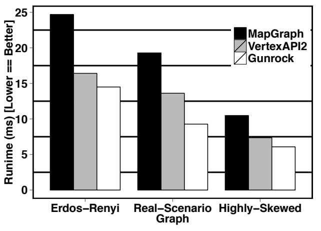


(a) Random Regular Graphs

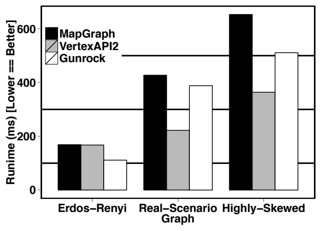


(b) Scale-free R-MAT Graphs

Fig. 5: Impact of average degree on performance.



(a) Traversal-based: BFS



(b) Computation-based: PageRank

Fig. 6: BFS and PageRank performance on different scale-free graphs with the same number of vertices and edges.

social-network-like graphs but graphs with other behaviors as well. We set the ratio of the parameters a, b, and c (choosing b = c for symmetry) to 8 (highly-skewed scale-free graph), 3 (closest to many real-world scenarios) and 1 (Erdös Rényi model). The result is three synthetic graphs with the same vertex and edge count but different skewness. Fig. 6 shows our runtime results for BFS graph traversal and dense-computation-based PageRank. On both 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 graph, and the highly-skewed graphs have the smallest diameter. On the other hand, for 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: 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 for all 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 graph topology than computation-based primitives, because they operate on a smaller subset of the graphs and hence have less parallel work to do per operation. Next, we turn away from *how* our graph analytic frameworks perform on different topologies to the underlying reasons *why* they perform that way.

C. *Operator Load Balancing Strategies*

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

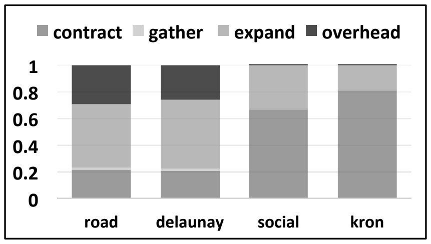
Load Balancing Methods: The graph analytic frameworks we study in this work use three distinct techniques to achieve parallel workload mapping at runtime:

1) When the frontier is so small that there is no way to fully utilize the GPU, load balance 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.
2) One alternative is Merrill et al.'s dynamic workload mapping [9] (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. This strategy achieves good utilization within blocks; however, it still potentially suffers from intra-block imbalance.
3) The other alternative is Davidson et al.'s partitioned load-balancing workload mapping [8] (PLB), which ignores any difference in neighbor list size and always chooses to map a fixed amount of workload to one block. When the frontier size is small, it maps a fixed number of vertices to a block. All threads expand all the neighbor lists cooperatively. When the frontier size is large, it maps a fixed number of edges to a block. To make all threads cooperatively visit all edges and know to which vertex's neighbor list each edge belongs require extra work, either an extra load-balanced search or a sorted search.

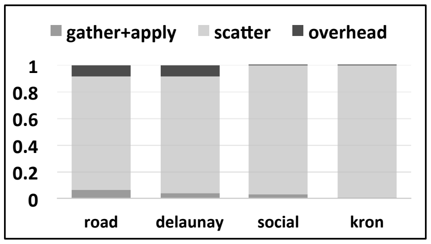
VA uses PT when frontier size is small and PLB when large. MG uses PLB for gather and dynamically switches between DWM and PLB (two-phase) for scatter according to frontier size. GR also dynamically chooses between DWM and PLB, not by frontier size but instead according to the graph type (DWM for mesh-like graphs and PLB for scale-free graphs). Both MG and VA implement PLB using load-balanced search while GR implements PLB using sorted search.

In Figs. 3 and 4, we see significant performance differences between VA and MG, despite both using the same GAS programming model. MG's two-phase method is efficient in expending the small frontiers that commonly appear in long-diameter graphs. VA's PT strategy hurts its performance when the frontier size is small, but it spends less time on redundant removal on scale-free graphs. MG behaves similar to GR since similar dynamic strategies are used in both frameworks. However, GR'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 that are common for scale-free graphs. A successful framework must carefully and dynamically consider both graph topology and frontier size to pick the best mapping strategy.

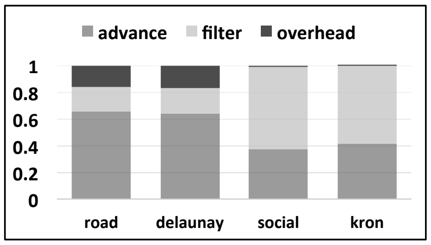
Fig. 7 provides a level breakdown for BFS: in which



(a) MG



(b) VA



(c) GR

Fig. 7: Phase breakdown of BFS. In MG, the expand phase enumerates neighbors and the contract phase generates next-level vertices. VA uses apply to update labels and scatter to activate the next level. In GR, filter is used to generate the new frontier while updates can be either in advance or filter, depending on mode. Overhead encompasses synchronization and data movement.

Primitive	Framework	road	delaunay	social	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 II: Average warp execution efficiency (W_{EE}).

stages 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 long-diameter road networks, which require more bulk synchronized super-steps, have much more synchronization overhead; their execution time is mostly occupied traversing neighbor lists. Conversely, scale-free and social networks introduce significant redundant edge discovery, thus contract/filter operations dominate their execution time.

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 II shows the average W_{EE} of different graph primitives for three frameworks. For BFS, social and kron graphs enable a higher average warp execution efficiency across all frameworks due to the highly-optimized load-balanced graph traversal operators used by each framework. However, 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 primitives with dense computation such as PageRank, all three frameworks achieve very high W_{EE} because all vertices in the neighbor lists actively participate in 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 a higher W_{EE}.

D. Synchronizations and Kernel Execution Patterns

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 super-step, and 2) implicit global synchronizations between GPU kernel invocations within each super-step. The BSP barrier count is directly proportional to the iterations required for a primitive to converge, and kernel invocation count corresponds to the number of synchronizations within each iteration. Each kernel invocation performs four steps: read data from global memory, compute, write results, and synchronize. Many graphs with long tails have substantial synchronization overhead.

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[2]. Table III summarizes the barrier count B_C and kernel count K_C. Both B_C and K_C show strong positive correlations with achieved performance as in Table IV. For traversal-based primitives where all three frameworks follow similar approaches, fewer synchronizations (implying kernels that do more work) yield superior performance: expert programmers fuse kernels together to reduce 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 calls. Generally, high-level programming models trade off increased kernel invocation overhead (compared to hardwired implementations) for flexibility and more diverse expressivity. That being said, reducing kernel invocations (K_C) is a worthwhile goal for any framework implementation, and GR's ability to fuse computation stages into advance or filter kernels

[2]The huge performance gap between GR and the GAS implementations on CC is primarily from GR's ability to run Greiner's PRAM-based CC algorithm [26], 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, especially for long-tail graphs.

		BSP Barrier Count (B_C)				Kernel Invocations Count (K_C)			
Dataset	Primitive	MapGraph	VertexAPI2	Gunrock	Hardwired	MapGraph	VertexAPI2	Gunrock	Hardwired
road_usa	BFS	6,263	6,263	6,263	6,262	**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**	—
delaunay_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-LiveJournal1	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 III: BSP barrier count (B_C) and kernel count (K_C) of BFS, SSSP, CC, and PageRank for each framework vs. hardwired implementations of BFS: b40c [9], and CC: Soman et al. [11]. Bold indicates fewest among the three programmable 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 IV: Correlation of B_C and K_C with performance (measured as throughput). PageRank B_C data is not applicable because of its fixed iteration count.

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 V: Global atomic transactions in each framework.

appears to directly translate into fewer kernel calls and thus higher performance.

E. Atomic Operation and Memory Impact

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.

Atomic Operation: Atomic instructions are generally considered expensive, although their cost has decreased with more recent GPU micro-architectures. Unfortunately, they are a key ingredient in operations on irregular graph data structures, due to the large amount of concurrent discovery, particularly characteristic of scale-free graphs. Table V summarizes each framework's number of global atomic transactions. BFS/SSSP's atomics are found in MG's expand/contract, VA's activate, and GR's filter. MG and GR show similar atomic behavior because they visit neighbors and do push-updates, then contract/filter out redundant vertices; for both, the work of contract/filter is substantial. VA uses more atomic operations on road networks for BFS/SSSP because its simple atomic activates (per-thread-expand) are frequently used; conversely, its fewer atomics in scale-free graphs are due to VA's use of a Boolean flag to indicate vertex status in the new frontier-generating phase.

GR results in this paper incorporate an idempotent mode of operation, 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 MTEPS from 891.7 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, this optimization is not applicable because GAS frameworks cannot guarantee the idempotence of arbitrary user-defined apply functions [13]. Turning to PageRank, we note that PageRank's runtime is dominated by vertex-centric updates and ideally suited to the GAS abstraction; neither VA nor MG requires atomics in their implementations, and partially as a result, both frameworks deliver excellent performance.

Data Movement: Another metric directly proportional to kernel invocations of data-intensive graph primitives is the data movement across GPU kernels. Let the number of vertices be v and the number of edges e, and consider BFS in both programming models: The GAS model requires $5e + 6v$ data transfers of which $4e + 3v$ are coalesced. In the data-centric model, both of GR's load balancing strategies require $5e + 4v$ global data transfers, of which $2e + 3v$ are uncoalesced.

Dataset	MapGraph		VertexAPI2		Gunrock	
	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 VI: Achieved global load and store transactions on BFS. Delanuay_n17 contains 131k vertices and 393k edges while kron_g500-logn17 has 131k vertices and 10.1M edges.

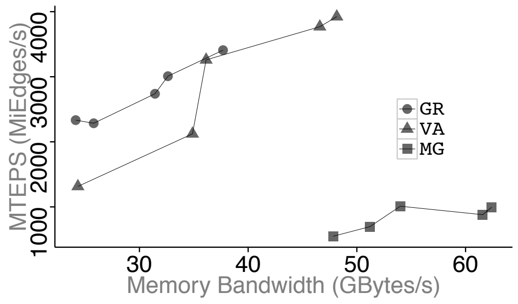
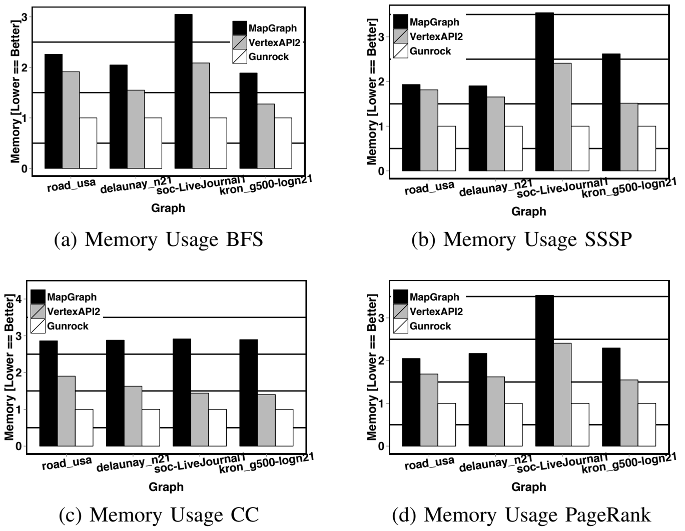


Fig. 8: Global memory bandwidth impact on BFS MTEPS.



(a) Memory Usage BFS (b) Memory Usage SSSP

(c) Memory Usage CC (d) Memory Usage PageRank

Fig. 9: Memory usage comparisons for BFS, SSSP, CC, and PageRank across four datasets, normalized to GR = 1.

Table VI summarizes the actual global load and global store counts for the three frameworks running BFS on two types of graphs. It supports the above theoretical analysis. We notice that for delaunay_n17, MG requires more data communication than GR and VA. Potential reasons for this behavior are: 1) the cost of maintaining an edge frontier between MG'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 data to do computation. In contrast, GR's PLB only has one level of such loads. VA 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, MG and GR both have 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 has the best performance.

Memory Usage and Bandwidth: All frameworks use the Compressed Sparse Row (CSR) format to store graphs. CSR is a good fit for its space efficiency and its ability to easily strip out adjacency lists and find offsets using parallel-friendly primitives [9]. CSR contains a row pointer array and column indices array. Compress Sparse Column (CSC) is similar to CSR except it stores column pointers and row indices. 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.

Without losing generality, Fig. 8 illustrates traversal throughput (MTEPS) as a function of memory bandwidth running BFS on 5 kron datasets (logn17 - logn21). All three frameworks follow the same pattern: higher achieved memory bandwidth leads to higher throughput. VA and GR 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 or gather + apply + scatter supersteps. In the data-centric model, the advance uses CSR to expand the neighbors of a current vertex frontier. In GAS, the gather phase requires a CSC to gather its neighbors (pull), and the scatter phase requires CSR (push) to complete push-style updates/activations. Thus the GAS implementations require storing the topology in both CSR and CSC formats. The usage of CSC doubles the memory usage of GAS compared to GR for directed inputs. However, GR integrates optimizations that consume memory beyond only CSR, such as direction-optimized traversal ("pull") on BFS, which requires a CSC input. GR's pull-enabled BFS requires 3.52 GB on kron_g500-logn21 (1.26X compared with non-pull-enabled). This optimization improves the traversal rate from 3476.6 to 8384.5 MiEdges/s. Here, GR demonstrates a trade-off between memory usage and performance.

Fig. 9 shows the global memory usage comparisons. 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, MG uses a two-step procedure to build CSR and CSC formats in memory, which results in an overall factor of 1.88X∼3.54X above GR. VA's implementation eliminates the scatter phase, which results in less usage than MG (1.27X∼2.4X compared to GR). The memory footprint for GAS implementations is clearly not optimal. Ultimately the larger memory usage of the GAS implementations limits the size of graphs that can fit into the GPU's limited on-board memory (12 GB on the K40c).

VII. Conclusion

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 for primitives dominated by vertex-centric updates; however, it suffers from slow information propagation and high memory usage. A data-centric abstraction enables more complex and powerful operators and lower memory usage. In practice, GR's implementation allows integrating more work in each kernel, thus requiring fewer kernel invocations and synchronizations 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 primitives beneath a high-level abstraction 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.

None of the programmable frameworks show convincing performance 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 common to all three frameworks. Using an asynchronous execution framework may be an interesting direction for future work. A second challenge is automatic kernel fusion, which can potentially reduce synchronization cost, but current GPU programming frameworks do not perform this (difficult) optimization automatically. Finally, memory performance is a crucial aspect of any GPU graph primitive and a fruitful area for future study. All three frameworks share the CSR graph format, but alternate graph formats might allow superior memory performance, particularly with regard to data coalescing.

Acknowledgments

Thanks to our funding agencies DARPA (XDATA award W911QX-12-C-0059 and STTR award D14PC00023) and NSF (awards OCI-1032859 and CCF-1017399), and to our DARPA program managers Wade Shen and Christopher White. We also appreciate the contributions and feedback from our Onu/Royal Caliber colleagues Vishal Vaidyanathan and Erich Elsen.

References

[1] 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*, OSDI '12. USENIX Association, Oct. 2012, pp. 17–30.

[2] 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*, UAI-10, Jul. 2010, pp. 340–349.

[3] 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 *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, Jun. 2010, pp. 135–146.

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

[5] 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*, PPoPP '13, Feb. 2013, pp. 135–146.

[6] 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*, GRADES '14, Jun. 2014, pp. 3:1–3:6.

[7] 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.

[8] 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*, IPDPS 2014, May 2014, pp. 349–359.

[9] 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*, PPoPP '12, Feb. 2012, pp. 117–128.

[10] 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.

[11] 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*, IPDPSW 2010, Apr. 2010, pp. 1–8.

[12] 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.

[13] 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.

[14] 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*, GRADES '14, Jun. 2014, pp. 2:1–2:6.

[15] 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*, HPDC '14, Jun. 2014, pp. 239–252.

[16] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *IEEE International Symposium on Workload Characterization*, IISWC-2014, Oct. 2014, pp. 140–149.

[17] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *IEEE International Symposium on Workload Characterization*, IISWC-2013, Sep. 2013, pp. 185–195.

[18] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *IEEE International Symposium on Workload Characterization*, IISWC-2014, Oct. 2014, pp. 130–139.

[19] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE International Symposium on Workload Characterization*, IISWC 2012, Nov. 2012, pp. 141–151.

[20] 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*, HPDC '15, 2015, pp. 173–178.

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

[22] 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*, PPoPP 2015, Feb. 2015, pp. 271–272.

[23] 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.

[24] J. H. Kim and V. H. Vu, "Generating random regular graphs," in *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, Jun. 2003, pp. 213–222.

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

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