Hindawi
Scientific Programming
Volume 2018, Article ID 9340697, 11 pages
https://doi.org/10.1155/2018/9340697

Research Article

HPGraph: High-Performance Graph Analytics with Productivity on the GPU

Haoduo Yang,[1,2] Huayou Su,[1,2] Qiang Lan,[1,2] Mei Wen,[1,2] and Chunyuan Zhang[1,2]

[1]*Department of Computer, National University of Defense Technology, Changsha 410000, China*
[2]*National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410000, China*

Correspondence should be addressed to Haoduo Yang; 18976522633@163.com

Received 7 July 2018; Accepted 31 October 2018; Published 11 December 2018

Academic Editor: Can Özturan

The growing use of graph in many fields has sparked a broad interest in developing high-level graph analytics programs. Existing GPU implementations have limited performance with compromising on productivity. HPGraph, our high-performance bulk-synchronous graph analytics framework based on the GPU, provides an abstraction focused on mapping vertex programs to generalized sparse matrix operations on GPU as the backend. HPGraph strikes a balance between performance and productivity by coupling high-performance GPU computing primitives and optimization strategies with a high-level programming model for users to implement various graph algorithms with relatively little effort. We evaluate the performance of HPGraph for four graph primitives (BFS, SSSP, PageRank, and TC). Our experiments show that HPGraph matches or even exceeds the performance of high-performance GPU graph libraries such as MapGraph, nvGraph, and Gunrock. HPGraph also runs significantly faster than advanced CPU graph libraries.

1. Introduction

Graph computing has become critical for analyzing data in many domains, such as in bioinformatics, social networking, web analysis, and traffic engineering. During the past decade, in terms of dealing with large-scale graphs, various parallel graph computing frameworks have been proposed for leveraging modern massively parallel processors, specifically graphics processing units (GPUs). GPUs which have excellent peak throughput and energy efficiency [1] have demonstrated very strong computational performance with appropriate optimization. However, the unpredictable control flows and memory divergence on GPUs caused by the irregularity of graph topologies need sophisticated strategies to ensure efficiency, making an efficient implementation on GPUs a challenge. With graphs getting larger and queries getting more complex, it is imperative for high-level graph analysis frameworks to help users extract the information they need with minimal programming effort.

In order to bridge the gap between high performance and productivity, we propose HPGraph, a high-level parallel graph analytics framework on GPU. The key abstraction of our framework is mapping vertex programs to generalized sparse matrix vector multiplication (SPMV) operations by CUDA kernels. Unlike other GPU graph computing models which focus on sequencing the steps of computation [2], we instead convert graph traversal to matrix operations so that we can focus on manipulating on data structures and provide high performance brought about by the optimized generalized SPMV. In addition, HPGraph encapsulates the complexity of programming and achieves the high productivity by hiding the underlying matrix primitives for users. Users with limited knowledge of low-level GPU architectures are therefore able to assemble complex graph primitives.

Our contributions to this field are as follows:

(1) We propose an efficient graph analytics framework which maps vertex programs to generalized sparse matrix operation on the GPU. This abstraction,

unlike the abstractions of pervious GPU graph procecssing libraries, is able to develop a wide range of graph primitives while simultaneously delivering high performance.

(2) HPGraph is productive for users. We hide some of the more unsavory facts of how HPGraph is really getting the job done, and provides a set of flexible APIs to express several graph primitives at a high level of abstraction.

(3) HPGraph integrates a variety of GPU-specific optimization strategies for data structures, graph traversal, and memory access into its core to further improve performance. In our experiments, our graph primitives significantly outperform other advanced CPU graph analytics frameworks and achieve comparable or even superior performance to other state-of-the-art GPU analytics libraries.

(4) We provide a detailed experimental evaluation of our graph primitives including comparisons to the performance of several advanced CPU and GPU implementations.

The remainder of this paper is organized as follows: Section 2 presents the existing graph frameworks and the motivations for our work. Section 3 describes our implementation and optimizations in detail. Section 4 discusses the implementation details of the graph algorithms. Section 5 provides the results of measuring the performance of the frameworks. In Section 6, we conclude the paper and discuss potential areas for future research.

2. Related Work and Motivation

This section will discuss the research landscape of large-scale graph analytics frameworks, which differ both in terms of programming models as well as the supported platforms.

Parallel graph analytics frameworks propose various high-level programmable models, such as vertex programming, matrix operations, tasks models, and declarative programming. Among them, vertex programming is quite widely applied and is generally productive for writing graph programs. However, since it focuses on sequencing the steps of computation and lacks a strong mathematical model, it is difficult to analyze due to its long runtimes and high memory consumption [3]. On the contrary, matrix models are built with a solid mathematical background, e.g., graph traversal computations are modeled as operations on a semiring in CombBLAS [4] and nvGRAPH [5]. This is beneficial in reasoning and performing optimizations, but it is considered difficult to program [6].

Single-node CPU-based systems are in common use for graph computation. Giraph [7] uses an iterative vertex-centric programming model similar to Google's Pregel [8], and it is based on Apache Hadoop's MapReduce. PowerGraph [9], designed for real-world graphs which have a skewed power-law degree distribution, uses the more flexible Gather-Apply-Scatter abstraction. These methods partition edges across nodes with a vertex-cut which exposes greater parallelism in natural graphs. Galois [10–12] is one of the highest performance graph systems for shared memory adopting a task-based abstraction. CombBLAS [4] and GraphMat [3] are two popular matrix programming models. CombBLAS is an extensible distributed-memory parallel graph library offering a small but powerful set of linear algebra primitives specifically targeting graph analytics. GraphMat, developed by Intel, is a single-node multicore graph framework mapping Pregel-like vertex programs to high-performance sparse matrix operations. Recent work [3] and [13] have compared different graph frameworks on CPUs. These papers show that GraphMat significantly outperforms many other frameworks in most cases. Moreover, the capability of mapping many diverse graph operations to a small set of matrix operations provides considerable convenience for the backend of GraphMat to maintain and extend itself, for example, to multiple nodes.

GPUs are power-efficient and able to carry out high-memory-bandwidth processing. They can exploit parallelism in computationally demanding applications. Most high-level GPU programming models for analytics today mirror CPU programming models. For instance, Zhong and He introduced Medusa [14], a high-level GPU-based system for parallel graph computing using Pregel's message model [8]. VertexAPI2 [15], MapGraph [16], and CuSha [17, 18] adopt PowerGraph's GAS programming model [9]. Gunrock [2] is a more recent library for developing graph algorithms on a single GPU. Rather than designing an abstraction around computation, Gunrock instead uses a GPU-specific data-centric model centered on operations on a vertex or edge frontier. Wang et al. [2] report that, compared to hardwired GPU implementations, Gunrock has comparable performance to the fastest GPU hardwired implementations and achieves better performance than any other GPU high-level graph library. nvGRAPH (nvGRAPH is available at https://developer.nvidia.com/nvgraph) is a high-performance GPU graph analytics library developed by NVIDIA. It harnesses the power of GPUs for linear algebra and matrix computations to handle the large-scale graph analytics [5]. The core functionality is using semi-ring SPMV operations to express graph computation [2]. It currently supports three algorithms: PageRank, SSSP, and Single-Source Widest Path.

Compared to CPU graph frameworks, existing high-level GPU graph frameworks usually gain improved performance due to their strengths in terms of hardware, the generalized load balance strategies, and optimized GPU primitives. Nevertheless, the unpredictable control flows and memory divergence on GPUs caused by irregular graph topologies need sophisticated strategies to ensure efficiency. This can result in relatively low productivity and high memory consumption.

Some matrix-based frameworks on CPUs, e.g., CombBLAS [4], GraphMat [3], and PEGASUS [19], have proven that a vertex-based programming model on CPUs can be established with a matrix backend for graph programming. Meanwhile, GPUs have the potential to accelerate sparse matrix algebra due to their memory-bound nature. A variety of optimizations have been performed to improve the performance of SPMV [20], one of the most important operations in high-performance computing (HPC), on GPUs. However, as far as we know, existing matrix-based graph analytics on GPUs achieve nowhere near the same performance as these optimized libraries [21, 22]. In this work, our goal is to achieve high performance (optimized sparse matrix backend) for graph analytics as well as the productivity of vertex programming (such as vertex programming for users) for GPUs.

3. The HPGraph's Abstraction and Optimizations

3.1. HPGraph's Abstraction. HPGraph is based on the idea that traversals from a vertex can be expressed as an operation which is similar to dot product, an element of SPMV routines on the graph adjacency matrix (or its transpose). Hence, HPGraph maps graph analytics using vertex programming to generalized SPMV on the GPU to deliver high performance. It targets graph operations which can be expressed as iterative convergent processes. By "iterative," we refer to operations which may require running a series of steps repeatedly, and the results of one iteration are used as the starting point for the next iteration. By "convergent," we mean that the correct answer can be obtained with sufficient accuracy by these iterations before terminating the process.

HPGraph uses a bulk-synchronous model (BSP) [23]. In BSP, parallel programs are executed in synchronous phases, known as supersteps. Such operations are sufficient for portability and efficiency on the GPUs. Each iteration is a superstep in HPGraph. Our abstraction differs from other frameworks, particularly other GPU-based frameworks. Rather than focusing on sequencing the steps of computation, we focus on mapping vertex programs to manipulate data structures. The graph primitives we describe in this paper mainly include three steps: *PREPROCESS*, *SPMV*, and *APPLY*.

PREPROCESS: in the *PREPROCESS* phase, the graph is converted to an adjacency matrix which is stored in the GPU memory. Furthermore, according to the specific requirements of graph algorithms, HPGraph generates different property data for vertices and configures the framework parameters. In terms of data structures, HPGraph represents all per-node data as structure-of-array (SOA) data structures, which allow for coalesced memory accesses with minimal memory divergence.

SPMV: similar to Giraph [7], HPGraph marks some vertices (or a single vertex) as having an "*active*" status. In each iteration, each vertex only visits and interacts with its "*active*" neighbors. Supposing that G is a M-by-N sparse matrix storing the graph, x is a vector storing user-defined node properties. In SPMV phase, graph traversal is completed by generalized SPMV: $y = G * x$ (or $y = G^T * x$). The vector y stores the promising new properties of each node, which will be used in APPLY phase. The corresponding operations on the sparse matrix are based on the idea that visiting the adjacent vertices can be performed through a dot product which is described as follows: If a vertex r visits one of its "*active*" neighbors, l, along out-edges ($r, l \in [0, M]$), a function named "*gather*" will be executed using $x[l]$, $G[r][l]$, and the properties of these two vertices. Conversely, visiting along in-edges requires us to perform a transposition firstly and obtain matrix G^T. Therefore, $x[l]$ and $G^T[r][l]$ can be used directly. The "reduce" function will summarize a new property using the results from "gather" operations and store it in the resultant array y. The above process can be substituted by a dot product in generalized SPMV.

Figure 1 shows a simple example of calculating out-degrees. A native SPMV operation, which uses an adjacency matrix converted from the graph and a vector of all ones as an input, can produce the out-degrees of all vertices stored in a vector. Concretely, a vertex visits along the out-edges with multiplication (i.e., "gather" operation) and adds (i.e., "reduce" operation) together to obtain its out-degree.

Our abstraction is sufficient to express a large number of diverse graph algorithms efficiently. Compared to other SPMV-based GPU graph libraries such as nvGraph, HPGraph provides convenient expression to algorithms like Triangle Counting with access to the properties of vertices in SPMV. On the contrary, such an approach can remove the need for some atomic operations in a parallel implementation than other graph-centric abstractions such as Gunrock and MapGraph. In other GPU programmable graph frameworks, there could be multiple parallel writers for the same vertex. This process needs atomic operations to ensure mutual exclusion. In our abstraction, writing data for an element in the output array is mapped to reduction for a row using a thread or a segment. With this mapping in place, HPGraph removes most of the contention, and the graph traversal and computation are simple and intuitively described. In general, HPGraph can achieve efficient implementation using a high-performance approach for SPMV.

APPLY: based on programmer-specified criteria, an *APPLY* step uses the resultant array y from the generalized SPMV to update the state of the vertices. Meanwhile, it sets those vertices, whose status has been changed, as "*active*" vertices and removes those redundant vertices. HPGraph performs that operation in parallel across all elements. This parallel scan is regular and well-suited to GPUs.

The HPGraph computation pipeline is shown in Figure 2. HPGraph primitives are assembled from multiple iterations including a sequence of *SPMV* and *APPLY*. They are mainly executed sequentially: one step completes all of its operations before the next step begins. Typically, HPGraph graph primitives run to convergence, which usually equates to no state-changed vertices or no "*active*" vertices. Besides, programmers can also specify the maximum number of iterations. HPGraph will terminate when the number of iterations reaches the predefined limit.

3.2. Optimizations. Due to our focus on manipulating some data structures, such as the sparse matrix and vectors, it is easy to allow for integrating optimizations into our framework for giving more options to programmers. We offer four examples.

Sparse Matrix Format: In the *PREPROCESS* phase, a large number of graphs are converted into a sparse matrix. Unlike most other frameworks of graph computing, which use compressed sparse row (CSR) for graph storage or vertex operations, we adopt the HYB [24] matrix format to store the graph throughout the paper.

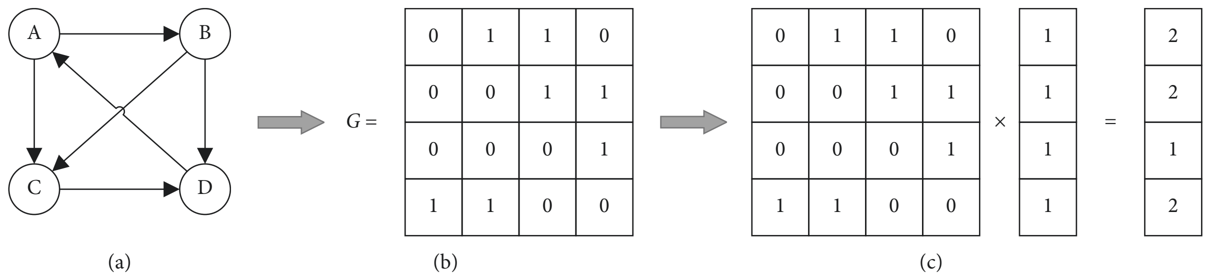

FIGURE 1: A simple example for computing out-degree with a SPMV operation. (a) Logical representation. (b) The adjacency matrix storage graph. (c) Out-degree calculation using SPMV $y = G * x$. The out-degrees are stored in the resulted y.

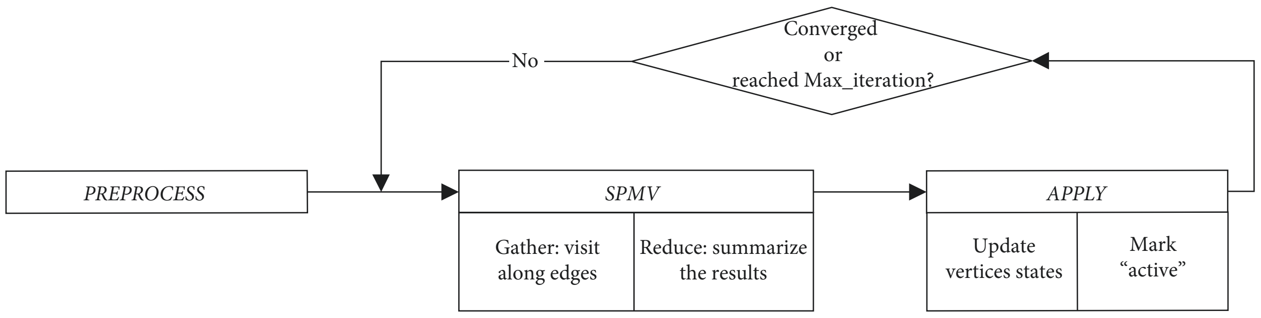

FIGURE 2: HPGraph computation pipeline.

For an $M * N$ matrix with a maximum of K nonzero elements per row, ELL (ELLPACK format) [25] uses a dense $N * K$ array data to store the nonzero values, where rows with less than K nonzeros are zero-padded. Similarly, the corresponding column in dices is stored in an index array, again with zero for padding. It is appropriate for a matrix with nonzero elements evenly distributed between rows. COO (coordinate format), a notably simple storage scheme, uses three arrays, row, col, and data, to store the row indices, column indices, and values of the nonzero elements in a matrix, respectively. In our implementation, the row array is sorted, ensuring that entries with the same row indices are stored contiguously.

The ELL format is well-suited for vector architectures. However, when the number of nonzeros per row varies widely, there will be a larger number of zero elements which should be filled. This can result in a rapid decline in efficiency. Meanwhile, the required storage for the COO format is always proportional to the number of nonzeros. Furthermore, this characteristic (the similar properties) extends to the cost of an SPMV using COO [26]. HYB, a hybrid ELL/COO storage format, stores some discrete matrix entries in COO and the majority entries using ELL. Given a sparse matrix A, N is the number of rows in A and NNZ is the number of nonzero values in A. Define a threshold K, and the part exceeding K nonzeros in a row is extracted and stored by COO. And, the remaining part is placed into ELL in order to minimize zero-padding. The sparse matrix can be divided into two parts. One contains the rows with less than K nonzeros, and the other contains the rows with more than K nonzeros. The first part is placed into ELL. For the second part, only K nonzeros in front of those rows with more than K nonzero elements are placed into ELL. The remaining entries of the second part are extracted for storage COO. Assuming that $K = 2$, Figure 3 illustrates the HYB representation of an example matrix. Bell et al. mentioned in [24] that, based on empirical results, it is beneficial to add Kth column to the ELL if at least one-third of the matrix rows contain K (or more) nonzeros. We adopt this idea to achieve the optimal threshold K.

For many (but not all) large graphs, the nonzero values of the graph adjacency matrix (or its transpose) are mostly concentrated on a small area. This is appropriate for HYB format. At present, HYB has been implemented on GPUs such as for the functions developed by NVIDIA CUDA Sparse Matrix library (cuSPARSE) [27] and CUSP [28] for SPMV, but it has not been implemented for graph analyses on GPU platforms. Based on the SPMV functions for the HYB-formatted matrix in the CUSP [28] library, we propose two kernels, executing operations for two submatrices, respectively.

Bottom-Up Traversal: Scott et al. [29] described a bottom-up traversal for BFS on CPUs. Instead of each vertex in the "*active*" frontier attempting to become the parent of all its neighbors, each unvisited vertex attempts to find any parent among its neighbors, which can reduce the total number of edges examined. This approach is beneficial when the number of unvisited vertices drops below the size of the visited vertices. Furthermore, HPGraph adapts this and uses a dot product to find these nodes' predecessors. Specifically, in the ith row of a transpose matrix A^T storing a graph, a nonzero value represents an edge pointing to node i. If $A^T[i][j]$ is valid and node j is a member of the visited set, node j can be a parent of i, and we do not need to check the rest elements in

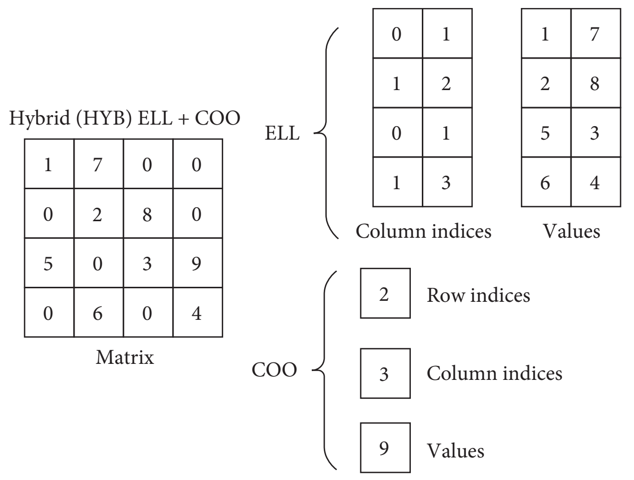

Figure 3: An example of the HYB format (assuming $K = 2$).

the same row. Meanwhile, any operation for processing the ith row can be skipped if vertex i has been already visited. Currently, in HPGraph, this optimization is only applied to BFS. In future, it could fit other graph algorithms as well.

Memory Access Optimization: in HPGraph, our strategy for active-status-lookup incorporates a bitmask to reduce the size of status data from a 32-bit label to a single bit per vertex, which can reduce the overhead of inspecting a given vertex's visitation status [30]. We create two bitmasks, named *active_bitmask* and *x_bitmask*, respectively. In *PREPROCESS* phase, *active_bitmask* is initialized to 0, while *x_bitmask* is initialized based on the algorithms used. In each iteration, the SPMV kernels access the array *x_bitmask* through the texture cache for improving performance. SPMV uses *x_bitmask* to specify the states of vertices. Once any vertices changed state, the array *active_bitmask* is atomically updated to ensure correctness. Then, the *APPLY* step updates *x_bitmask* using *active_bitmask* and resets *active_bitmasks* before next iteration.

Furthermore, CUDA devices have several kinds of addressable memory (i.e., registers, shared memory, constant memory, texture memory, local memory, and global memory). As illustrated in [24], SPMV operations on GPUs are highly memory latency bound. The indirect memory access pattern easily leads to statically unpredictable memory access behavior which is dependent on the input graphs. In view of this, we utilize the following methods to optimize device memory access.

We propose to load the read-only data of the HYB-formatted sparse matrix via the read-only cache on Kepler, which can benefit the performance of kernels with limited bandwidth. In terms of the input arrays of the *SPMV* phase, which stores vertices states and some bitmasks, they are read-only for the entire lifetime of SPMV kernels as well. Due to the frequent and random access to these arrays in SPMV kernels, we have decided to store them in the texture memory, which can improve performance and reduce memory traffic.

In general, compared to other GPU computation-focused abstractions, HPGraph has been an excellent fit for these alternatives and optimizations, especially for SPMV.

3.3. Programming Interface. In order to facilitate the implementation of a wide range of graph primitives, HPGraph provides a series of simple and flexible application programming interfaces (APIs) (Figure 4). A set of configuration parameters, as the library calls for iteration control and other functionalities, is initialized in the *PREPROCESS* phase. The user-defined functions will then be integrated into *SPMV* or *APPLY* kernels calls. In terms of hiding any complexities concerning how these steps are internally implemented, our proposed framework achieves high programmability and productivity. Users need to write only C++ code to use the framework.

4. Applications

One of the principal advantages of HPGraph's abstraction is that *SPMV* and *APPLY* can be composed to build various graph primitives with minimal unnecessary work. We describe below how we express each primitive in HPGraph.

4.1. Breadth-First Search (BFS). BFS is one of the most important graph algorithms and often used in conventional searches, such as web crawling. It aims to explore all the vertices connected from the source. It starts from a given source and iteratively expands to find all the reachable vertices. In HPGraph, the properties for each vertex include "depth," which shows the minimum number of edges that need to be traversed from the source to a particular vertex and "parent" which shows the predecessor vertex's ID. One can think of the algorithm as performing the following computation once per vertex per iteration:

$$\text{depth}(i) = \min_{j|(j,i)\in E} \text{depth}(j) + 1. \tag{1}$$

BFS initialized all depths to infinity, except the starting vertex, which is set to 0. We implement a more efficient "bottom-up" traversal mode (see Section 3.2 for details) for reducing the concurrent discovery of child nodes. In BFS, since any race condition between edges visiting a vertex is benign, HPGraph only requires flags for visited and unvisited vertices. When a vertex's predecessor is discovered, we set its predecessor ID in the *SPMV* phase and its depth in the *APPLY* phase. Nodes are never revisited in a later iteration. So, by recording the vertices visited in every iteration, it is easi to get the actual path finally.

4.2. Single-Source Shortest Path (SSSP). In a directed and weighted graph with an appointed source vertex, SSSP computes the shortest path and its lengths between source and destination vertices. Many efficient solutions for this problem have been put forward, while we use a method which is a slight variation on the Bellman–Ford shortest path algorithm used in the LonestarGPU graph library [31]. Compared to BFS, SSSP maintains distances, representing the minimum edge weights needed to reach a vertex from the source, rather than depth. If a new distance is updated, this vertex may be revisited in a later iteration. Therefore, SSSP necessitates recording edge weights, the distances of

Parameters	Edges_type	Visit along in-edges, out-edges, or all-edges
	Max_iteration	The max number of iterations to perform
	Node_property	The algorithm-specific property data of vertices
Functions	Gather	Operations in *SPMV* phase for interacting with "*active*" neighbors
	Reduce	Operations in *SPMV* phase for producing a resulted vector
	Update	Operations in *APPLY* phase for updating states of vertices

Figure 4: HPGraph's API set.

all nodes. Algorithm 1 describes in detail how it maps to HPGraph's abstraction. In our proposed abstraction, the distance of the source vertex is initialized as having 0, while all the other vertices are initialized to a ∞ value. HPGraph accesses all associated in-edges for each vertex and uses the "*active*" ones (whose distances are changed in the last iteration) attached to those edges to relax the distance's value (if necessary). We use an atomic operation to retain a bitmask associated with the updated vertices to remove the redundant vertices. The same process repeats until convergence is achieved.

4.3. PageRank (PR). PageRank is an algorithm which is used to rank web pages according to their popularity for estimating the importance of them. This application iteratively computes the page rank of every vertex in a directed graph. The ranks of the pages are recursively influenced through the hyperlinks and are updated by the following equation in each iteration until convergence is achieved:

$$\mathrm{PR}(v) = d + (1-d) * \sum_{u\mid(u,v)\in E} \frac{\mathrm{PR}(u)}{C(u)}, \qquad (2)$$

where E is the set of edges in a directed graph, $\mathrm{PR}(v)$ is the page rank of page v, $\mathrm{PR}(u)$ is the page rank of page u which links to page v, $C(u)$ is the number of links on page u, and d is a damping factor which can be set between 0 and 1. Furthermore, we set the initial rank of every vertex to 1.0. HPGraph computing the out-degree of each vertex in the PREPROCESS phase. All vertices are initialized as "*active*" in the graph. The *SPMV* phase calculates the sum of the weighted received across in-edges, and the remainder is completed in *APPLY* phase. We use a bitmask array to remove the vertices whose page ranks have already achieved convergence.

4.4. Triangle Counting (TC). Triangle Counting in graphs is a statistics algorithm which is frequently used in complex network analysis, random graph models, and important real-world applications such as spam detection, uncovering the hidden thematic structures in the Web and link recommendation [32]. In HPGraph, we view TC problem as a set intersection problem based on the idea that a triangle exists when a vertex has two adjacent vertices that are adjacent to each other [3]. We change the input graph to be directed acyclic for TC. For a given directed graph with no cycles, the size of the intersections gives the number of triangles in the graph. Specifically, let the intersections between the neighbor lists of u and v be $(w_1, w_2, w_3, w_4, \ldots, w_N,)$, where N is the number of intersections. Then, the number of triangles formed with e is N, where the three edges of each triangle are (u, v), (w_i, u), and (w_i, v) $(i \in [1, N])$. In general, the TC algorithm in HPGraph has two stages: firstly, HPGraph forms neighbor lists for all vertices; secondly, it computes the size of set intersections.

5. Experiments and Results

5.1. Experimental Setup. Experimental Platform: we ran all experiments described this paper on a Linux workstation with two Intel(R) Xeon(R) CPU E5-2620 CPUs, each with 6 cores running at 2.40 GHZ. And, we conduct our GPU experiments, complied with CUDA Toolkit 8.0, on a NVIDIA Tesla K40m GPU. K40m is equipped with 2,880 stream cores, 12 GB on-board memory, and memory bandwidth up to 288 GB/sec. The GraphMat was compiled using the Intel ICPC 17.0.4 compiler. All tests were executed 10 times with the average runtime used for results. We report the time taken to run the graph algorithms after loading the graph into the GPU or CPU memory (excluding the time taken to allocating resources or reading the graph from disk). Meanwhile, we show the millions traversed edges per second (MTEPS) for both BFS and SSSP.

Datasets: Table 1 provides the base characteristics of the datasets used in our experiments. The chosen datasets include both real-world and generated graphs, and the topology of these datasets spans from regular to scale-free. Data on LiveJournal1, Youtube, Pokec, and Orkut are collected from social networks; Kron_g500-logn21 is a generated R-MAT graph. They are all scale-free graphs with diameters of less than 25 and unevenly distributed node degrees, while roadNet-CA has relatively larger diameters with small and evenly distributed node degrees [33]. MapGraph, nvGRAPH, and Galois have not done the work for TC. And, although Gunrock has mentioned its experiment for TC on [34], its open-source repository does not include this part. So, we merely compare the HPGraph's performance of TC to that of GraphMat. The edge weight

```
procedure PREPROCESS $G^T$, $P$, $active\_bitmask$, $root$
    P.distance $[1..G^T$.vertices$] \longleftarrow \infty$
    P.distance[root] $\longleftarrow 0$
    reset_vertex_active(1..$G^T$.vertices)
    set_vertex_active(root, active_bitmask)
end procedure

procedure Generalized_SPMV $G^T$, $P$, $y$, active_bitmask
    for $v = 1 \longrightarrow G^T$.vertices do
        temp $\longleftarrow$ P.distance[$v$]
        for $u = 1$ in $G^T$.column$_v$ do
            if check_vertex_active(u, active_bitmask) then
                temp $\longleftarrow$ Gather(temp, P.distance[$u$], $G^T[u, v]$)
            end if
        end for
        $y \longleftarrow$ Reduce(P.distance[$v$], temp)
    end for
end procedure

procedure Gather temp, $a$, $b$
    return min(temp, $a + b$)
end procedure

procedure Reduce $a$, $b$
    return min($a$, $b$)
end procedure

procedure APPLY $P$, $y$, active_bitmask
    for $v = 1 \longrightarrow G^T$.vertices do
        reset_vertex_active($v$, active_bitmask)
        if update($y$, P. distance[$v$]) then
            P.distance[$v$] = $y$
            set_vertex_active($v$, active_bitmask)
        end if
    end for
end procedure

procedure Update $a$, $b$
    return ($a < b$)
end procedure
```

Algorithm 1: Single-Source Shortest Path, expressed in HPGraph's abstraction (G is the graph and P is the properties of vertices).

values (used in SSSP) for each dataset are random values between 1 and 64.

5.2. Performance Results

5.2.1. HPGraph vs. GPU Graph Libraries. For BFS, SSSP, and PageRank, we compared the performance of HPGraph to that of three state-of-the-art high-level GPU graph analytics frameworks: MapGraph, Gunrock, and nvGRAPH. The comparison results are shown in Table 2.

The geometric mean values of HPGraph's accelerations over MapGraph on BFS, SSSP, and PageRank are 2.4, 2.8, and 1.8, respectively. For Gunrock, they are 1.3, 1.1, and 6.4, respectively. From Table 2, it is clear that HPGraph's performance is comparable or even superior to that of MapGraph and Gunrock on BFS and SSSP. However, we noticed that for roadNet-CA, HPGraph is about 1.7x slower than MapGraph and 3.0x slower than Gunrock. This performance inconsistency is due in part to the issue that BFS or SSSP on the small-degree large-diameter graphs requires taking a lot of iterations to finish with each iteration merely updating a limited number of vertices. For example, we need over 500 iterations before reaching converged in roadNet-CA. As a result, HPGraph, which has a relatively bigger per-iteration overhead, performs poorly. In terms of PageRank, HPGraph significantly outperforms MapGraph and Gunrock. In general, HPGraph shows greater acceleration on PageRank, which has dense computation and more regular frontier structures [2] than traversal-based graph primitives (BFS and SSSP).

For SSSP, nvGRAPH is slower than HPGraph on all datasets (average of 4.3X). For PageRank, HPGraph achieves comparable performance with nvGRAPH (average of

Table 1: Dataset descriptions.

Dataset	Vertices (M)	Edges (M)	Max/avg. degree	Diameter
LiveJournal Davis and Hu [35]	4.8	68.9	2.7 k/14	16
Youtube Leskovec and Krevl [36]	1.1	2.7	29 k/3	20
Pokec Leskovec and Krevl [36]	1.6	30	14.9 k/27	11
sx-stackoverflow Leskovec and Krevl [36]	2.6	63.4	38 k/48	9
Orkut Rossi and Ahmed [37]	3	106	27 k/70	9
Kron_g500-logn21 Davis and Hu [35]	2	91	214 k/86	6
roadNet-CA Davis and Hu [35]	2	5.5	12 k/2	849

Table 2: Performance comparison (runtime and edge throughput) between GPU implementations.

Alg.	Dataset	Runtime (ms) (lower is better)				Edge throughput (MTEPS) (higher is better)			
		MapGraph	Gunrock	HPGraph	nvGRAPH	MapGraph	Gunrock	HPGraph	nvGRAPH
BFS	LiveJournal	111	**63**	68		622	**1095**	1015	
	Youtube	28	18	**13**		107	166	**230**	
	Pokec	35	32	**30**		875	957	**1021**	
	sx-stackoverflow	170	115	**34**		374	552	**1868**	
	Orkut	165	216	**120**		645	492	**886**	
	Kron_g500-logn21	133	**18**	27		685	**5058**	3372	
	roadNet-CA	180	**69**	173		31	**80**	32	
SSSP	LiveJournal	280	**71**	122	348	246	**972**	566	198
	Youtube	40	16	**13**	59	75	187	**230**	51
	Pokec	128	**39**	45	165	239	**785**	681	186
	sx-stackoverflow	280	131	**58**	180	227	485	**1095**	353
	Orkut	540	230	**135**	920	197	462	**788**	116
	Kron_g500-logn21	38	**19**	26	201	2396	**4742**	3502	379
	roadNet-CA	110	**64**	186	351	50	**86**	30	12
PageRank	LiveJournal	52	55	28	**20**				
	Youtube	4	50	**2**	4				
	Pokec	34	60	**13**	14				
	sx-stackoverflow	63	63	23	**18**				
	Orkut	**29**	174	42	30				
	Kron_g500-logn21	61	165	34	**20**				
	roadNet-CA	5	7	**4**	5				

All PageRank times are normalized to one iteration. Best results for each example are shown in bold.

1.0X). Specifically, nvGRAPH is faster than HPGraph on four datasets and slower on three (Youtube, Pokec, and roadNet).

5.2.2. HPGraph vs. CPU Graph Libraries. We compared the performance of HPGraph to that of two advanced CPU graph libraries: Galois [10–12], a high-performance-task-based framework; GraphMat [3], a high-performance matrix programming framework. Both use the entire system (24 cores). As Figure 5 shows, HPGraph is significantly faster than both Galois and GraphMat. Compared to Galois, HPGraph's performance is generally better in most of the tested cases, and it achieves 4.1-6.3x speedup over the range of algorithms and datasets. While against GraphMat, HPGraph provides 4.6x speedup on all primitives (average of 3.4x for BFS, 5.4x for SSSP, 5.1x for PageRank, and 3.2x for TC).

5.3. Effect of Optimizations. Figure 6 shows the performance effect of our optimizations (described in Section 3) for BFS (running on LiveJournal) and PageRank (running on Kron_g500-logn21). We use an implementation with CSR-based generalized SPMV as the baseline naïve version. The first bar shows the baseline naïve implementation normalized to 1. Adding bitmask arrays to mark "active" vertices and memory access optimization provides few benefits as shown in the second bar. This enables better parallel scalability. The third bar indicates that the HYB-formatted sparse matrix results in a further gain of 2.5x for BFS and 1.95x for Pagerank. The direction-optimal traversal strategy (mentioned in Section 3.2) also works better for BFS. Similar results were obtained for other algorithms and datasets as well. In general, our flexible GPU-specific programming model, which allows for integrating efficient optimizations, including faster matrix operations and more effective graph traversal, can make HPGraph productive without sacrificing any performance.

5.4. Discussion of Performance. These two GPU BFS-based high-level-programming-model efforts (both MapGraph and Gunrock) used for comparison adopt load-balancing

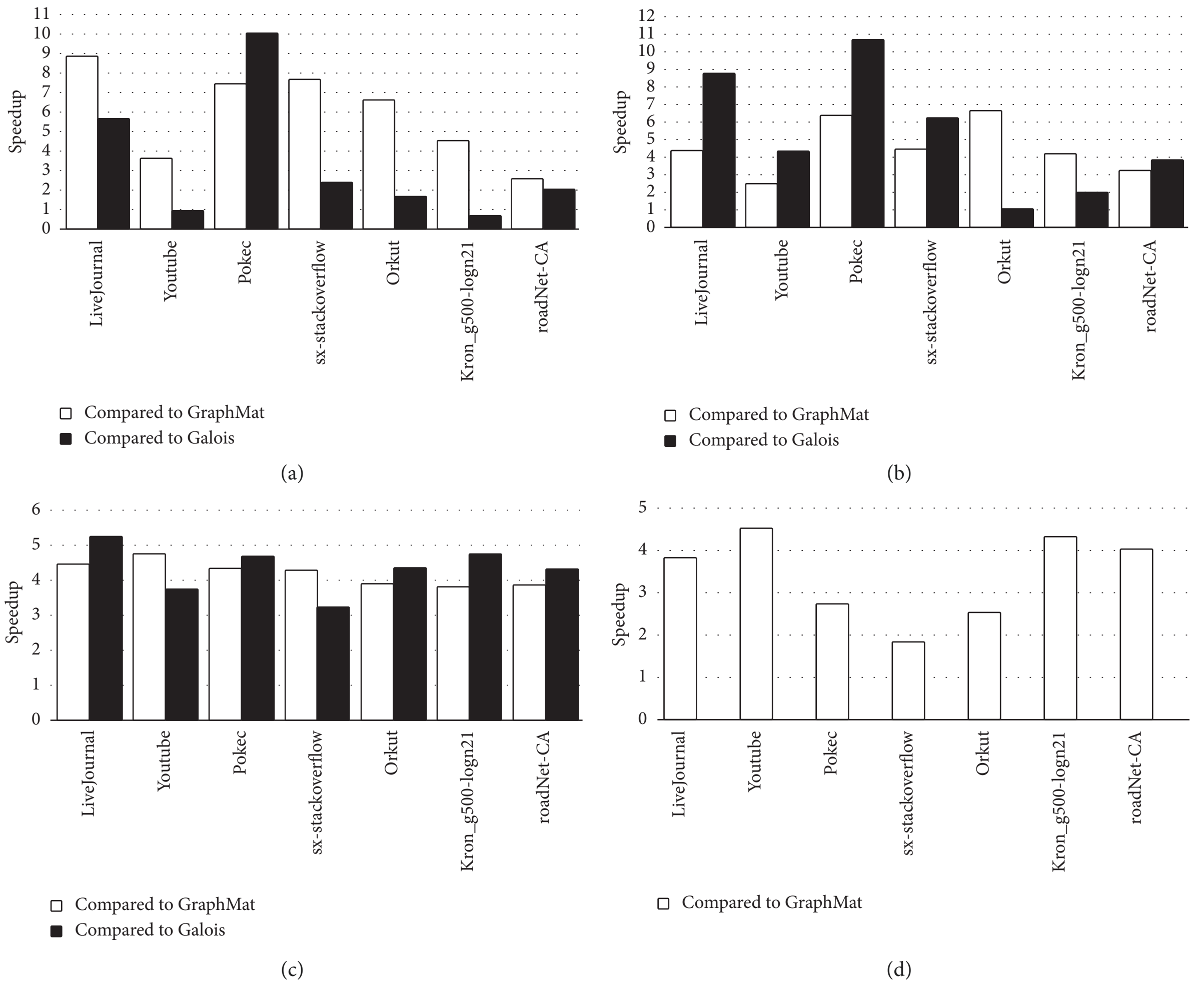

Figure 5: Summary of the performance improvement of HPGraph over GraphMat and Galois on four algorithms. (a) BFS. (b) SSP. (c) PageRank. (d) TC.

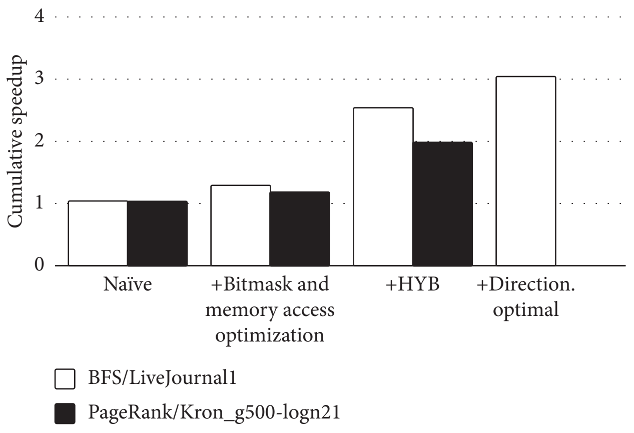

Figure 6: The effect of optimizations performed on naive implementations of BFS and PageRank algorithms (direction-optimal strategy is only used for BFS).

strategies from Merrill et al.'s BFS [38]. Without its optimization methods, we still expect HPGraph to show similar performance on all graph primitives as these other frameworks. With mapping the vertex programs to generalized sparse matrix operations, HPGraph supports better sparse matrix format, efficient graph traversal, and memory access optimizations. All these reasons result in the superior performance on average warp efficiency of HPGraph's SPMV kernels across all algorithms and datasets (shown in Table 3). Average warp efficiency means the fraction of threads active during computation, which is a good measure of load-balancing quality.

On the contrary, nvGRAPH uses a similar matrix backend as HPGraph, but HPGraph still outperforms nvGRAPH especially in SSSP. The primary cause for this is that we have heavily optimized our generalized SPMV backend as described in Section 3. Since nvGRAPH is a closed-source, a detailed comparison is infeasible. Furthermore, as shown in [6], some graph computations, such as Triangle Counting, are hard to express efficiently as a pure matrix operation which leads to long runtimes and increased memory consumption. That is probably the main reason why nvGraph has not done the work for Triangle Counting. With allowing the properties of vertices to be visited during SPMV, HPGraph completes efficient implementation for Triangle Counting and shows better productivity.

Table 3: Average warp execution efficiency, which is a good metric for the quality of a framework's load-balancing capability.

Datasets	BFS (%)	SSSP (%)	PageRank (%)
LiveJournal	97.81	87.52	99.52
Youtube	95.32	87.4	99.63
Pokec	93.73	89.97	99.42
sx-stackoverflow	99.25	85.45	99.53
Orkut	98.59	87.67	99.56
Kron_g500-logn21	99.87	89.8	99.4
roadNet-CA	85.98	89.62	99.49

6. Conclusions

Graph analytics have been widely applied in many applications [39, 40]. In this paper, we have presented HPGraph, a GPU graph analytics framework which maps a vertex programming to an optimized matrix backend. Such a model provides both high performance and productivity. Since the main computational method is based on SPMV, HPGraph can achieve improved performance using modified high-performance SPMV primitives. Furthermore, HPGraph's abstraction has allowed us to integrate various optimization strategies including data structures, direction-optimal traversal, and memory access. Meanwhile, the high-level abstraction allows users to complete various graph primitives with little effort. The experimental results on large-scale graphs show that HPGraph runs significantly faster than the advanced CPU library and can match or even exceed the performance of other state-of-the-art programmable GPU graph libraries. Since the available GPU memory is still the bottleneck for bigger datasets in HPGraph, in future work, we expect the framework to scale well from a single GPU version to multi-GPU clusters.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors gratefully acknowledge supports from the National Key Research and Development Program under No. 2016YFB1000400 and the National Nature Science Foundation of China under NSFC Nos. 61502509 and 61402504.

References

[1] 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, 2011.

[2] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, p. 11, Barcelona, Spain, March 2016.

[3] S. Narayanan, N. Satish, M. M. A. Patwary et al., "Graphmat: high performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[4] A. Azad, A. Buluc, and J. R. Gilbert, *Combinatorial Blass v1.6.1*, EECS, Berkeley, CA, USA, 2018, https://people.eecs.berkeley.edu/aydin/combblas/html/index.html.

[5] J. Kepner, A. Peter, D. Bader et al., "Mathematical foundations of the GraphBLAS," in *Proceedings of 2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, Waltham, MA, USA, September 2016.

[6] N. Satish, S. Narayanan, M. M. A. Patwary et al., "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 979–990, Chicago, IL USA, May 2014.

[7] C. Avery, "Giraph: large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.

[8] G. Malewicz, M. H. Austern, J. C. B. Aart et al., "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 135–146, Indianapolis, IN, USA, June 2010.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: distributed graph-parallel computation on natural graphs," *OSDI*, vol. 12, no. 2, 2012.

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

[11] The University of Texas, *Galois v2.2.1*, The University of Texas, Austin, TX, USA, 2014, http://iss.ices.utexas.edu/?p=projects/galois/download.

[12] K. Pingali, D. Nguyen, M. Kulkarni et al., "The tao of parallelism in algorithms," *ACM Sigplan Notices*, vol. 46, pp. 12–25, 2011.

[13] A. Iosup, Tim Hegeman, W. L. Ngai et al., "LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms," *PVLDB*, vol. 9, no. 13, p. 12, 2016.

[14] 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, 2014.

[15] E. Elsen and V. Vaidyanathan, *A Vertex-Centric CUDA/C++ API for Large Graph Analytics on GPUs Using the Gather-Apply-Scatter Abstraction*, Github, San Francisco, CA, USA, 2013.

[16] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: a high level API for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pp. 1–6, Snowbird, UT, USA, June 2014.

[17] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 39–50, San Francisco, CA USA, October 2015.

[18] 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*, pp. 239–252, Vancouver, BC, Canada, June 2014.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “Pegasus: a peta-scale graph mining system implementation and observations,” in *Proceedings of Ninth IEEE International Conference on Data Mining ICDM’09*, pp. 229–238, Miami, FL, USA, 2009.

[20] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, “Fast sparse matrix-vector multiplication on GPUs for graph applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 781–792, New Orleans, LA, USA, November 2014.

[21] V. Kalavri, V. Vlassov, and S. Haridi, “High-level programming abstractions for distributed graph processing,” *IEEE Transactions on Knowledge & Data Engineering*, vol. 30, no. 2, pp. 305–324, 2018.

[22] H. Yang, H. Su, M. Wen, and C. Zhang, “HPGA: A High-Performance Graph Analytics Framework on the GPU,” in *Proceedings of 2018 International Conference on Information and Computer Technologies*, pp. 584–588, Qingdao, China, July 2018.

[23] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[24] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on CUDA,” NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Santa Clara, CA, USA, 2008.

[25] J. R. Rice, “ELLPACK: progress and plans,” in *Elliptic Problem Solvers*, pp. 135–162, Academic Press, Cambridge, MA, USA, 1981.

[26] M. M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies,” IBM Reserach Report RC24704, IBM, Yorktown Heights, NY, USA, 2008.

[27] NVIDIA, *The NVIDIA Cuda Sparse Matrix Library*, NIVIDIA, Santa Clara, CA, USA, 2015, https://developer.nvidia.com/cusparse.

[28] S. Dalton, N. Bell, L. Olson, and M. Garland, *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*, NVIDIA Corporation, Santa Clara, CA, USA, 2014, http://cusplibrary.github.io/version0.5.0.

[29] B. Scott, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[30] H. Yang, H. Su, Q. Lan, M. Wen, and C. Zhang, “High performance graph analytics with productivity on hybrid CPU-GPU platforms,” in *Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications*, pp. 17–21, Hong Kong, China, 2018.

[31] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in *Proceedings of 2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 141–151, La Jolla, CA, USA, November 2012.

[32] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, “Efficient triangle counting in large graphs via degree-based vertex partitioning,” *Internet Mathematics*, vol. 8, no. 1-2, pp. 161–185, 2012.

[33] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters,” *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[34] Y. Wang, Y. Pan, A. Davidson et al., “Gunrock: GPU graph analytics,” 2017, http://arxiv.org/abs/1701.01170.

[35] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[36] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, Stanford University, Stanford, CA, USA, 2014, http://snap.stanford.edu/data.

[37] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, TX, USA, January 2015, http://networkrepository.com.

[38] M. Adam 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*, pp. 572–583, New Orleans, LA, USA, November 2014.

[39] X. Chen, P. Li, J. Fang, T. Tang, Z. Wang, and C. Yang, “Efficient and high-quality sparse graph coloring on GPUs,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4064, 2017.

[40] H. Zhang, X. Chen, N. Xiao et al., “Shielding STT-RAM based register files on GPUs against read disturbance,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 2, p. 17, 2017.