Graph Processing on GPUs: A Survey

XUANHUA SHI and ZHIGAO ZHENG, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
YONGLUAN ZHOU, Department of Computer Science, University of Copenhagen, Denmark
HAI JIN, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
LIGANG HE, Department of Computer Science, University of Warwick, United Kingdom
BO LIU and QIANG-SHENG HUA, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China

In the big data era, much real-world data can be naturally represented as graphs. Consequently, many application domains can be modeled as graph processing. Graph processing, especially the processing of the large-scale graphs with the number of vertices and edges in the order of billions or even hundreds of billions, has attracted much attention in both industry and academia. It still remains a great challenge to process such large-scale graphs. Researchers have been seeking for new possible solutions. Because of the massive degree of parallelism and the high memory access bandwidth in GPU, utilizing GPU to accelerate graph processing proves to be a promising solution. This article surveys the key issues of graph processing on GPUs, including data layout, memory access pattern, workload mapping, and specific GPU programming. In this article, we summarize the state-of-the-art research on GPU-based graph processing, analyze the existing challenges in detail, and explore the research opportunities for the future.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; • **Computing methodologies → Massively parallel algorithms**; • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Parallel computing models**;

Additional Key Words and Phrases: Graph processing, GPU, graph datasets, parallelism, BSP model, GAS model

ACM Reference format:
Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6, Article 81 (January 2018), 35 pages.
https://doi.org/10.1145/3128571

This work is partly supported by three grants from the NSFC (No. 61370104, No. 61433019, and No. U1435217), a grant from the International Science and Technology Cooperation Program of China (No. 2015DFE12860), and a grant from the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032).
Authors' addresses: X. Shi, Z. Zheng, H. Jin, B. Liu, and Q.-S. Hua, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China; emails: {xhshi, zhengzhigao, hjin, borenaliu, qshua}@hust.edu.cn; Y. Zhou, Department of Computer Science, University of Copenhagen, Universitetsparken 5, DK-2100, Copenhagen, Denmark; email: zhou@di.ku.dk; L. He, Department of Computer Science, University of Warwick, Coventry, CV4 7AL, United Kingdom; email: Ligang.He@warwick.ac.uk.

https://doi.org/10.1145/3128571

1 INTRODUCTION

A graph is a mathematical structure that consists of a set of vertices and edges connecting certain pairs of them (Bondy and Murty 1976). Much real-world data can be naturally represented as graphs, and therefore the concept of graphs has been applied to various applications, where the relationships among objects play an important role. Below are some examples of real-world graph applications:

- —In chemistry, graphs are widely used to model the molecule structures, where the vertices and the edges represent atoms and the chemical bonds between them. Such graph representation of the molecular structures forms the basis of building the software for searching molecules.
- —In physics, graph theory is widely used in the study of three-dimensional structures of atoms, where each vertex stands for an atom and an edge connects a pair of atoms if there is interaction between them. The edges are weighted by the interaction strength between two vertices. Such a graph model provides an intuitive representation that facilitates the research of atomic structures.
- —In computational neuroscience, graphs are used to represent the functional connections between brain areas that interact with each other in various cognitive processes. In such graph models, the vertices and edges represent different brain areas and their connections, respectively.
- —In social sciences, graphs are also widely used, for example, for the social network analysis. The relationship among people can be naturally modeled as graphs, where an edge between two persons means they know each other and the edge weight indicates the influence of their relationship or the frequency of their interactions. Researchers can then extract interesting information from such graphs, such as measuring the actors' prestige (Polites and Watson 2008), exploring the way of rumor spreading (Azad et al. 2015), and so on.
- —In the study of the World Wide Web, researchers use directed graphs to represent the linked structure of web pages in the whole web, where a vertex represents a web page and a directed edge stands for the referencing relation between two web pages.
- —In computational linguistics, it has been proved that graph models are particularly useful in natural language processing (NLP), information retrieval, web link predictions, and many other applications. For instance, syntax and compositional semantics are often represented as tree-based structures, which greatly facilitate the formulation of the analysis tasks and is hence widely used in many NLP systems, such as CoreNLP (Manning et al. 2014), TextGraphs (Hahn and Reimer 1984), WordNet (Miller 1995), and so on.
- —In addition, graphs are also used to abstract and represent various structures in computer systems, such as computation flows, data organizations, and so on (WU et al. 2015). For example, in compiler optimization, graphs are often used to express the code structures, where vertices and edges represent functions (or classes) and function call relationships, respectively.

Given the wide applicability of graph models, developing graph analytic algorithms to explore and discover the underlying knowledge within graphs has been of great interest for a very long time (Lee and Messerschmitt 1987; Hall et al. 2009; Jordan and Mitchell 2015). However, the rapidly growing sizes of real-world graphs calls for new technologies to support the analysis of very large-scale graphs. For example, there are 342 million active users on Twitter,[1] and the Word Wide Web

[1] http://www.statisticbrain.com/twitter-statistics/.

graph contains more than 4.75 billion pages and 1 trillion URLs.[2] To address the challenge of scalability, the researchers have been making extensive efforts in developing scalable graph traversal algorithms, such as BFS (Liu and Huang 2015; Liu et al. 2016), and iterative graph analysis algorithms, such as PageRank (Mitliagkas et al. 2015; Richardson and Domingos 2001). To facilitate the development of arbitrary large-scale graph analysis applications, researchers have also developed generic graph programming frameworks both in the context of a single machine such as GraphChi (Kyrola et al. 2012), X-Stream (Roy et al. 2013), and GridGraph (Zhu et al. 2015), and in a cluster, such as Pregel (Malewicz et al. 2010) and PowerGraph (Gonzalez et al. 2012).

Recently, the technical advance of the General-Purpose Graphics Processing Units (GPGPUs) (Owens et al. 2007), especially the features of massive parallelism and high memory access bandwidth, has attracted a lot of researchers to investigate how to apply GPGPUs to accelerate computations in various applications including graph processing (Merrill et al. 2012; He et al. 2010; Li and Becchi 2013; Ashari et al. 2014). More recently, efforts have been devoted to building general graph-processing systems on GPUs, such as TOTEM (Gharaibeh et al. 2012), CuSha (Khorasani et al. 2014), GunRock (Wang et al. 2016), and Frog (Shi et al. 2015).

GPU adopts a SIMD-based (Single Instruction Multiple Data) architecture, which gains high performance through massive parallelism. In GPU, most of the die area is used by the Arithmetic Logic Units (ALUs), while a small proportion of the area is contributed to the control units and caches. Furthermore, GPU usually has a very high memory access bandwidth, but a limited memory space. This architecture enables GPU to perform regular computations in very large degree of parallelism (Colic et al. 2010; Lu et al. 2010).

On the contrary, modern multi-core CPUs adopt the MIMD (Multiple Instruction Multiple Data) architecture and the control units and caches take up most of the die area, with less remaining for ALUs. Comparing to GPUs, CPUs are better at performing tasks that demand short latency, which requires the support of complicated control units and large cache.

Although GPUs can offer a high degree of parallelism, their restrictions mean that it is a nontrivial task to use GPUs to accelerate large-scale graph computations. Graph computations often exhibit irregular data access patterns, due to which the applications may not reach the peak performance in GPU. Furthermore, due to the fact that the memory size of GPU is very limited compared with CPU memory and moving data from the host memory to the GPU memory causes the extra overhead, the GPU memory may become a potential bottleneck. In addition, the condition branches (e.g., the *if-else* statement) in graph computations do not fully exploit the high degree of parallelism offered by the SIMD executions in GPU, which leads to the so-called branch divergence and may dramatically degrade the performance. In this article, we attempt to write a comprehensive survey of the existing efforts in addressing these challenges, and at the same time discuss the main opportunities in graph processing on GPUs.

In order to cover the challenges of graph processing on GPUs, we surveyed about 100 papers published in recent years, as summarized in Table 1. The problems addressed in the existing research on graph processing on GPUs can be categorized into the following aspects:

—Data Layout. In conventional CPU-based graph-processing algorithms and systems, it is important to design a data layout to achieve contiguous memory access to enhance Translation Lookaside Buffer (TLB) and cache hit rates. But in a GPU, there is a global memory shared by all GPU processors, and for each memory access, it is beneficial to feed data to more than one SIMD thread. Threads in a GPU are executed in groups (a group is called a *warp* in Compute Unified Device Architecture (CUDA)). The accesses to the global memory

[2] http://www.worldwidewebsize.com/.

Table 1. Optimization Aspects on Graph Processing on GPUs

Aspects	Concerns	Related Work
Data Layout	regularity	CuSha (Khorasani et al. 2014), GStream (Seo et al. 2015), GTS (Kim et al. 2016)
	memory bandwidth	MapGraph (Fu et al. 2014), SPMV (Ashari et al. 2014), Frog (Shi et al. 2015)
	data-dependent parallelism	TOTEM (Gharaibeh et al. 2012), ExContract (Merrill et al. 2012), SBI (Brunie et al. 2012)
Memory Access Pattern	irregular memory access	In-Cache Query (He et al. 2014), TOTEM (Gharaibeh et al. 2012), Hybrid System (Abdullah et al. 2014), ExContract (Merrill et al. 2012), Enterprise (Liu and Huang 2015),
	non-coalesced memory access	In-Cache Query (He et al. 2014), Medusa (Zhong and He 2014), SPMV (Ashari et al. 2014), DWS (Meng et al. 2010), CuSha (Khorasani et al. 2014), MapGraph (Fu et al. 2014), GunRock (Wang et al. 2016), iBFS (Liu et al. 2016), Frog (Shi et al. 2015)
	bank conflict	DWS (Meng et al. 2010), Push-Relabel (Azad et al. 2015), WLP (Baghsorkhi et al. 2010)
	out-of-core processing	Warm-Up (Guha et al. 2015), GTS (Kim et al. 2016), Enterprise (Liu and Huang 2015), Green-Marl (Hong et al. 2012), PDOM (Fung et al. 2007), Frog (Shi et al. 2015)
	memory-dependent parallelism	GBTL-CUDA (Zhang et al. 2016),
	memory bandwidth	TOTEM (Gharaibeh et al. 2012), GTS (Kim et al. 2016), PDOM (Fung et al. 2007)
Workload Mapping	warp divergence	CuSha (Khorasani et al. 2014), DWS (Meng et al. 2010), iBFS (Liu et al. 2016), Virtual Warp (Hong et al. 2011a), PDOM (Fung et al. 2007), Two-Level Warp Scheduling (Narasiman et al. 2011)
	task scheduling	FinePar (Zhang et al. 2017), MapGraph (Fu et al. 2014), GunRock (Wang et al. 2016), TOTEM (Gharaibeh et al. 2012), Hybrid System (Abdullah et al. 2014), ExContract (Merrill et al. 2012), Enterprise (Liu and Huang 2015), SBI (Brunie et al. 2012)
Miscellaneous	branch divergence	DWS (Meng et al. 2010), Medusa (Zhong and He 2014), Virtual Warp (Hong et al. 2011a), PDOM (Fung et al. 2007), Two-Level Warp Scheduling (Narasiman et al. 2011), WLP (Baghsorkhi et al. 2010), SBI (Brunie et al. 2012)
	GPU specific programming	GunRock (Wang et al. 2016), Green-Marl (Hong et al. 2012), Medusa (Zhong and He 2014)
	other aspects	G2 (Zhong and He 2013), Mars (He et al. 2008; Fang et al. 2011), Morph (Nasre et al. 2013)

by the threads of a warp (or half a warp in older devices) will be coalesced into a single memory access if the consecutive threads are accessing the contiguous memory addresses. By doing so, the memory access overhead can be minimized. A GPU can reach its peak memory access bandwidth only when the algorithm has a regular memory access pattern, i.e., the data accessed by the consecutive threads of a warp occupies the contiguous memory segment. However, graph data structures and graph algorithms often issue irregular memory accesses. For instance, in a parallel graph traverse algorithm, when the adjacency-list data structure is used, different threads will access the data scattered across different memory locations, which requires the GPU to issue multiple memory accesses to fetch all needed data. Such irregular data layout substantially limits the degree of parallelism of a GPU and does not help unleash its full power. In addition, a GPU device typically communicates with the CPU host by a PCI Express (PCIe) bus or an Accelerated Graphics Port (AGP), which has a limited bandwidth. Therefore, it is critical to design the appropriate graph data layout to reduce the amount of data movements between the GPU and the host.

—Memory Access Pattern. CPU is usually equipped with a very large main memory, which is enough to process most real-world graphs. Furthermore, even with graphs that are larger than the main memory size, CPU-based systems can efficiently use secondary storage to handle the problem due to the relatively high bandwidth. But a GPU is usually equipped with high-speed but small-sized on-chip shared memory, which can be used to cache the frequently accessed data to reduce the need of accessing the on-device global memory. However, if many threads access different data in the shared memory concurrently, it will cause the conflicts of memory bank and hence limit the degree of parallelism. Furthermore, the access to the global memory in a modern GPU is usually in the unit of blocks. The block size is usually 64KB, but also depends on the GPU architecture. Therefore, if the accesses to global memory issued by a warp are coalesced and aligned within one or a few memory access units, then it can significantly improve the utilization of memory bandwidth. So similar to data layout, carefully designing the memory access pattern is also a critical issue in GPU computation.

Using the limited memory to process large-scale graphs that cannot fit into the GPU global memory, which is called out-of-core graph processing, is another major challenge (Kyrola et al. 2012; Roy et al. 2013; Khorasani et al. 2014). Partitioning graphs into small parts or designing smart graph data representations, with which the data are swapped in and out of the GPU memory when needed, may be the potential solutions to this problem. However, the research on how to organize the irregular graph data and the relevant performance study are relatively sparse. In addition, in this out-of-core graph-processing technique, whether the device memory is used or not can hugely influence the processing quality and power efficiency.

—Workload Mapping. CPU has a strong and flexible Control Unit, which can change the scheduling strategy flexibly in runtime. But GPU runs in a Single Instruction Multiple Threads (SIMTs) model. Once the instruction is distributed by the controller, it is impossible to change the scheduling strategy until the next iteration. Parallellizing graph computations often causes load imbalance due to the irregular graph structure. For instance, different vertices in a graph often have very diverse degrees, which complicates the balancing of the workload among the parallel tasks. An uneven load distribution among the threads within a kernel call may significantly harm the performance (Khayyat et al. 2013). Furthermore, as CPU and GPU favor different types of tasks, how to partition the workload between CPU and GPU so as to achieve good overall performance in such a hybrid system also becomes a challenging task.

—Miscellaneous. Besides the aforementioned aspects, implementing efficient graph computations on GPUs need to address various other issues, such as branch divergence, kernel calls, and kernel configuration. Thanks to its flexible Control Unit, CPU is good at handling condition branches. But for GPU, the branch divergence arises when different threads take different paths in a condition branch in the same wave-front. This will cause serious performance problems on GPUs, because only one path can be executed at a time in the SIMD mode of GPU, which means that only a portion of threads in a warp are running on a path while all other threads that should take other paths are blocked and not be able to perform any effective work (AMD 2011; Bienia and Li 2010; Meng et al. 2010). Avoiding branch divergence is a great challenge for GPU programmers. In addition, synchronous and asynchronous processing on GPU is another issue. As GPU is a parallel streaming processor, synchronous operations may limit the computing power of GPU. On the other hand, it is hard to implement asynchronous graph-processing operations on GPUs as there are a large number of messages passing between the vertices. Kernel configuration is a complex multidimensional structure, which reflects the hardware architecture of the GPU. In a parallel GPU programming, threads are grouped into the blocks for the convenience of inter-thread communication and memory sharing. A block can be of one-, two-, or three-dimensional structure; blocks can be further grouped into a grid. The grid is a one- or two-dimensional structure. Because GPU has the SIMD execution mode, each thread in the grid will compute the same kernel function on different parts of the same dataset. Thus, the kernel configuration has a significant effect on the degree of parallelism and hence influences the computing efficiency. A kernel is callable from the host while the kernel executes on the GPU device. Each thread is given a unique ID, which is generated when the kernel is invoked. As mentioned above, threads within the block share the same shared memory, through which they can cooperate with each other. Therefore, improper kernel invocations may cause the accessing conflict of memory banks and hence harm the computing power of GPU.

This survey focuses on graph processing on GPUs. In this survey, we group the GPU graph-processing systems into two categories, i.e., graph-processing systems on a single GPU and those on multi-GPUs. Accordingly, the remainder of this article is organized as follows. Section 2 presents some background information about graph processing on GPUs, such as CUDA, OpenCL, and GPU computing architecture. The implementation of graph algorithms on GPU is elaborated in Section 3. Section 4 introduces the GPU graph-processing systems, including both single GPU and multi-GPU graph-processing frameworks. Section 5 designs a series of experiments to show the performance with different data types and algorithms on GPUs. We conclude this article and discuss some research opportunities in Section 6.

2 BACKGROUND

Although the advent of general programming platforms and APIs, such as Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), have simplified the implementation of general computations in GPU, efficient GPGPU programming requires not only learning new GPU programming languages and APIs, but also understanding the underlying hardware and internal mechanisms in the GPU. This section presents the background information of the GPU architecture and two main GPU programming types: CUDA and OpenCL.

2.1 History and Evolution of GPU Architecture

The evolution of the modern graphics processor begins with the introduction of the first 3D add-in cards in 1995 (Seiler et al. 2008). From the perspective of parallel architecture, we can divide the evolution into three generations.

— **Fixed functional architecture.** From 1995 to 2000, each hardware unit consists of a graphics processing pipeline; the functions in the pipeline are fixed. In this generation, a plurality of pixel pipelines execute the same operation on each input data using the stream computing model. By using this architecture, GPU can significantly accelerate graphics rendering.
— **Separated shader architecture.** In 2001, NVIDIA's GeForce 3 introduced programmable pixel shading to the consumer market which sets off a new generation of GPU. In this generation, the programmable vertex shader replaces the illumination associated fixed units, and the pixel shader replaces the texture sampling and mixing associated fixed units. This greatly enhanced the flexibility and expressiveness of graphics processing. Although both of these parts are stream processors, they are physically separated and have no direct communication channel. Due to GPU's powerfulness in graphic rendering, it is widely used in gaming and other consumer applications.
— **Unified shader architecture.** The unified shader architecture emerged from 2006. In this generation, the geometry shader program was introduced in GPU, which can be dynamically scheduled to execute the vertex, geometry, and pixel programs. This generation of GPU adopts the parallel architecture rather than the streaming one. In addition, they support integer and single/double precision computations, and their instructions, textures, and data accuracy are further improved. However, they still cannot support recursive procedures. With the development of the computation power of GPU, GPGPUs emerged, which are not only for graphic shading, but also for high performance computing (HPC). Examples include the NVIDIA's Fermi, Kepler, Maxwell, and Pascal. Fermi was introduced in 2006, which is the first complete GPU computing architecture. In order to provide high accuracy computation for HPC, NVIDIA introduced the first Fermi-based product, GeForce 8800, in 2006 (NVIDIA 2009; Arjun et al. 2011), which is one of the most representative parallel computing processors. In 2012, NVIDIA introduced the Kepler architecture based on Fermi (NVIDIA 2012), which adopted some new features such as dynamic parallelism, Hyper-Q, grid management unit, and NVIDIA GPUDirect to provide higher processing power and parallel workload execution for HPC. With the focus on low power operations, NVIDIA proposed the Maxwell architecture in 2014 (NVIDIA 2014). In order to make the GPU more suitable for PCs, workstations, supercomputers and mobile chips, NVIDIA grouped streaming multiprocessors (SMs) into quads to minimize power consumptions. Since then, GPU is widely used in mobile chips. With the development of AI, Deep Learning, autonomous driving systems, and numerous other computing-intensive applications, NVIDIA introduced the Pascal architecture in 2016 (NVIDIA 2016b) to improve the support of computing-intensive applications with many new technologies including NVLink, HBM2 High-Speed GPU Memory, Unified Memory, Compute Preemption, and so on.

2.2 Modern GPU Computing Architecture

Figure 1 illustrates the overall architecture of a modern GPU device. To support massive parallel computing, a GPU typically consists of several SMs, each of which is composed of a number of GPU cores (alternatively called streaming processors), special functional units, registers, double-precision unit(s), and a thread scheduler. Each GPU core has the scalar integer and floating point arithmetic units, where most instructions of a GPU program are executed. A GPU core supports multithreading, typically supporting 32 to 96 threads in the current hardware.

The memory of a GPU device can be divided into two hierarchies: on-device memory and on-chip memory, which is shown in Figure 2. The on-device memory is the Dynamic Random Access Memory (DRAM), which is logically divided into local memory, global memory, constant memory, and texture memory; whereas, the on-chip memory consists of several physical components,

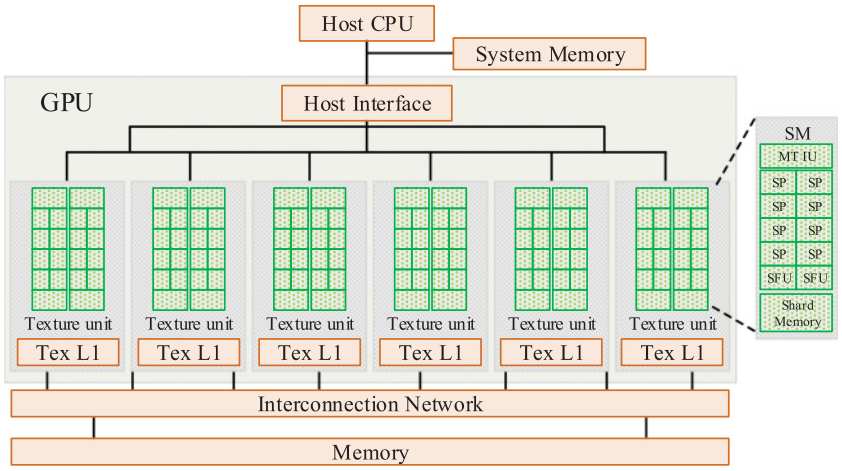


Fig. 1. GPU Architecture (NVIDIA 2016a).

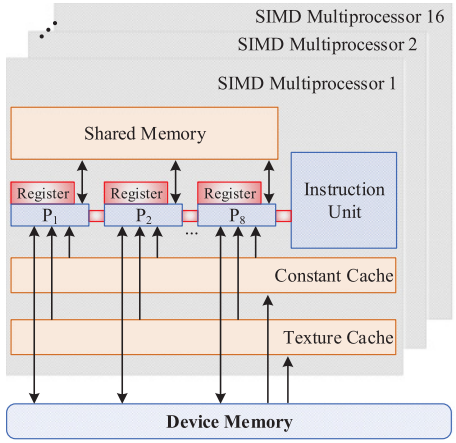


Fig. 2. GPU Memory architecture (NVIDIA 2016a).

including register shared memory, L1/L2 cache, and constant & texture cache. Each thread has access to a small and exclusive part of the on-device local memory, while all the threads can access the global memory. The logical constant and texture memory are mainly designed for graphical computations in image processing.

The access to the on-device memory usually has a long latency, in the unit of hundreds of clock cycles. Mainstream architectures include a two-level or three-level cache on a SM, including the constant and texture caches, and L1/L2 cache, to reduce the average memory latency. However, GPU is mostly used for stream computing, where cache is not important in general. (This is also why the die area contributed to the cache is very limited in GPU.) The shared memory on each SM is small (48KB per SM in NVIDIA K40 GPU[3]) but is of high speed, and is only accessible to the threads spawned on that SM.

[3]http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf.

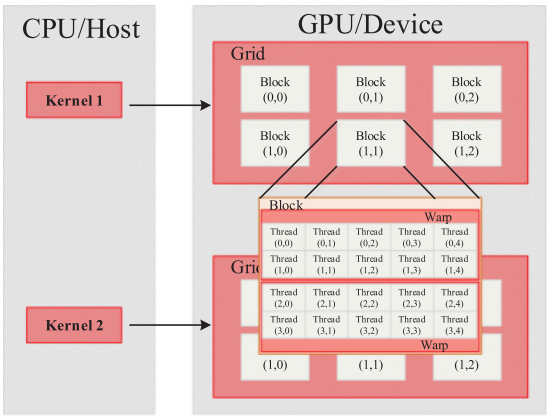


Fig. 3. CUDA programming model.

A GPU is generally used as a co-processor or an accelerator for the host CPU. A GPU is connected to the host by the PCI-Express bus. The data is transferred between the on-device memory in GPU and the main memory in CPU usually by using the programmed Direct Memory Access (DMA), which operates concurrently with both the host CPU and the GPU computing units. The zero-copy function is supported in some GPU architectures, such as CUDA from version 2.2 and OpenCL from version 1.2, where a GPU is able to access the host memory through PCIe and its on-device memory can be mapped into the host address space. This technique highly improves the communication efficiency.

2.3 CUDA

CUDA is probably the most popular general-purpose GPU programming framework, which is developed by NVIDIA. The CUDA architecture has a unified shader pipeline, allowing each Arithmetic Logic Unit (ALU) to perform general-purpose computations. There are three key concepts in CUDA: *thread hierarchy*, *shared memory*, and *barrier*.

Figure 3 shows the *thread hierarchy*. In CUDA, the first two layers of thread group is called warps and blocks. A programmer can set the number of threads per block subject to the hardware-dependent constraints, which is usually in the range of hundreds. All the threads in a block can access the *shared memory*, which works as a cache and can be used to share data among the threads within the block. A programmer can also set the number of threads per warp. CUDA can then divide the threads in a block into warps. The threads in a warp share the same code and follow the same execution path. During the computation, there is only one warp that can be executed at the same time in an SM, and all the warps mapped to an SM are executed in a time-sharing fashion. Each SM operates in a Single Instruction Multiple Threads (SIMT) fashion, where the SM issues one set of instructions to a warp of threads for concurrent executions over different data elements. Namely, the SM supports the instruction-level parallelism, but does not support branch prediction or speculative execution.

Finally, a number of blocks form a grid. A grid of threads execute the same GPU kernel, reading inputs from the global memory and writing the results back to the global memory. A device with capability 2.0 or higher can execute multiple kernels concurrently. The maximum number of threads allowed depends on the specific device capability. The kernels from different CUDA contexts cannot run concurrently. Different kernels synchronize only through the kernel calls.

Barrier is a thread synchronization construct, which is a point in the code where all the threads within a block synchronize. Only until all the threads have reached the barrier, can they proceed to execute the next instruction after the barrier. If, for some reason, some threads cannot reach the barrier, e.g., they are stuck in an infinite loop, then the threads that have reached the barrier will be blocked forever.

2.4 The OpenCL Programming Model

OpenCL is a more general programming framework than CUDA for heterogeneous architectures, which can be used on CPU, GPU, and some other processors or hardware accelerators, such as DSP and FPGA. Different from CUDA, which only supports data parallelism, OpenCL supports both task and data parallelism. Similar to CUDA, OpenCL also provides a general programming interface such as memory management, device management, kernel management, error checking, and information querying. Programmers can control the related device by using this interface.

OpenCL views the computing system as consisting of a number of computing devices, which can be CPU, GPU, or other accelerators. In OpenCL, a computing device contains several computing units and each computing unit is composed of multiple processing elements. Different from the memory hierarchy in CUDA, four types of memories are defined in OpenCL. The "global memory" is shared by all processing elements but with high latency. The "constant memory," which is small but with high speed, is writable only by the host CPU and read-only for other devices. The "local memory" is shared by a group of processing elements. Finally, the "private memory" (also called device register) is a fast on-chip memory.

3 GRAPH-PROCESSING ALGORITHMS ON GPU

One research direction in the literature is to study how to make use of GPUs' massive parallellism and high memory bandwidth to accelerate specific graph algorithms. Initially, the work of making use of GPU to accelerate specific graph algorithms mainly focused on graph traversal algorithms. More recent researches have studied more complicated algorithms, including Betweenness Centrality (BC), Connected Component (CC), Single Source Shortest Path (SSSP), PageRank (PR), and Minimum Spanning Tree (MST). This section attempts to discuss and summarize these existing efforts.

3.1 Traversal Algorithms

Traversal algorithms are a type of graph algorithms that visit each vertex of a graph in a certain pattern. Researchers have mainly studied how to efficiently perform Breadth-First Search (BFS) and SSSP on GPUs.

3.1.1 BFS. As one of the most important graph traversal algorithms, executing parallel BFSs on GPUs has attracted a lot of research efforts.

Memory Access Pattern. Merrill et al. (2012) adopt the Compressed Sparse Row (CSR) graph representation in their BFS traversal algorithm, which, as analyzed above, provides a compact and regular data layout.

However, the edge information needed by a warp may still not be coalesced and aligned in one memory access unit (i.e., 64 or 128B in modern GPUs). Therefore, the authors extract the information of the edges to be visited from the CSR representation, and then align them in one memory access unit. To avoid bank conflict of the shared memory, each thread broadcasts within the warp, the location of the shared memory it is going to access. Putting all these together, the authors reduce the complexity of the BFS algorithm to $O(|V| + |E|)$, while the methods proposed

by others have a quadratic complexity (Harish and Narayanan 2007; Hong et al. 2011b; Jia et al. 2011).

In order to process large-scale graphs, Liu and Huang (2015) proposed a GPU-based BFS framework, called Enterprise. In each iteration, Enterprise scans the status of the vertices and stores the status in a *Status Array*, and uses a *Frontier Queue* to store the unvisited adjacent vertex. As Enterprise executes the BFS in a tree manner, the unvisited adjacent vertices will be visited in the next level. Enterprise aligns the vertices in *Status Array* according to the *Frontier Queue*. By using this method, Enterprise can visit the memory in a regular fashion.

The parallel BFS proposed by Fu et al. (2014) extends the expand-contract BFS algorithm developed by Merrill et al. (2012) to GPU clusters. In their work, they propose a 2D partitioning method, and use Message Passing Interface (MPI) to contract columns on the edge frontiers after each expanding step. Their method has several disadvantages, such as algorithm generality, hardware compatibility, and scalability. The proposed parallel BFS method only works with the graph algorithms with no data access beyond direct neighbors, which limits the general applicability of the proposed method. Also, the proposed method limits the number of GPUs to n^2, in order to ensure the hardware compatibility of the algorithm. In addition, the proposed method ensures the scalability of the algorithm by reducing the edge frontier transmission between GPUs, which also reduces the communication overheads.

Job Mapping. In many graphs, the vertex degrees vary significantly, which causes load imbalance among the threads. To solve the problem, Hong et al. (2011a) propose to use the whole warp to explore the neighbors of a vertex (or a few vertices if the number of neighbors of one vertex is smaller than the number of threads in a warp), instead of using a single thread to explore all neighbors of a vertex. Besides being able to achieve load balance, this strategy also enhances the usage efficiency of the shared memory, because there are less data (i.e., less neighbor information) needed by the whole warp. Experiments show that small (sub-)graphs and the graphs with long diameters have poor performance on GPUs, but can archive good performance on multi-core CPUs. In order to utilize both multi-core CPU and GPU resources, Hong et al. (2011b) propose a hybrid scheme in which the graph is partitioned into several sub-graphs and the sub-graphs are distributed on the multi-core CPU and the GPU according to the number of vertices and the diameter of the graph. The partial results for the sub-graphs are combined to obtain the final result.

3.1.2 SSSP. SSSP is another typical graph traversal algorithm, which requires finding a shortest path between two specified vertices (Bulu et al. 2010).

Memory Access Pattern. Harish and Narayanan (2007) are the first to use CUDA to implement Dijkstra's algorithm, a traditional SSSP method. However, the implemented algorithm suffers from the inefficiency of atomic operations. By using the SSSP algorithm formulation, they also implemented the CUDA-based APSP (All-Pair-Shortest-Path) problem, which was originally solved by the Floyd-Warshall (FW) algorithm in CPU.

In the proposed CUDA-based APSP method, the global memory is used and the shared memory is not used because in APSP each thread can access the global memory, but finds it difficult to achieve data locality in the shared memory. This method is easy to use, but can hardly process large-scale graphs because of the limited device memory. The experiments on a NVIDIA GTX8800 GPU with the artificially generated high-degree graphs show that SSSP and APSP can achieve 70× and 17× speedup, compared with the performance of running the serial implementation on an Intel Core 2 Duo processor. However, in the experiments with real-life graphs that contain several millions of vertices, the method does not demonstrate a similar performance advantage. This is mainly due to the low average degree of these real-life graphs. Namely, the algorithms manifest a poor performance on the graphs with low degree. In order to solve this problem, a blocked FW algorithm

was proposed in 2008 by Katz and Kider (2008), which proposed a hierarchically parallel method in the revised FW algorithm. In this algorithm, the graph was represented by an adjacency matrix. In order to process large-scale graphs, this algorithm partitions the matrix into $B \times B$ equally sized sub-matrices. By using this method, the sub-matrices which have no relationship with each other can be calculated at the same time in the computation phase. In the first phase, the block $(0, 0)$ is loaded into the global memory for the computation. At the same time, the blocks $(0, i)$ ($i \neq 0$ and $i < B - 1$) and $(j, 0)$ ($j \neq 0$ and $j < B - 1$) are loaded into the shared memory. In the second phase, the blocks $(0, i)$ ($i \neq 0$ and $i < B - 1$) and $(j, 0)$ ($j \neq 0$ and $j < B - 1$) participate in the computation while block (1,1) is loaded into the shared memory, and so on. Benefiting from this shared memory strategy, the proposed method has a 5.0–6.5× speedup over Harish and Narayanan's work.

3.2 Iterative Algorithms

Iterative algorithms are very common in graph processing and machine learning. Many looping statements such as *while*, *loop*, or *do–while* are used in iterative algorithms. The algorithm executes the steps in iterations by using these looping statements. The aim of an iterative algorithm is to find the approximation solution by updating the vertex values successively.

PageRank. PageRank was first proposed by Google, and used in web link predictions. As the irregular memory access brought by the graph data, it is very hard to use GPU to process the PageRank. Rungsawang and Manaskasemsak (2012) implemented the PageRank on GPU by using the CSR representation. Wu et al. (2010) use a modified CSR format to represent the graphs. In order to solve the job mapping problem caused by uneven row sizes of the spare linkage matrices (degree of the vertex), Wu classifies the vertices into three classes, i.e., *Tiny Problems*, *Small Problems*, and *Normal Problems*, according to the amount of calculation. Wu assigns different numbers of threads to process the corresponding classes according to the computation task.

Sparse Matrix-vector Multiplication. Sparse matrix-vector multiplication (SpMV) is widely used in sparse linear algebra, and has been extensively studied. SpMV is a highly irregular computing algorithm. How to design a sufficient regular execution path and memory access pattern for SpMV is an interesting research topic. Filippone et al. (2017) surveyed the techniques for implementing SpMV on GPUs. The main issue of running the SpMV kernel on GPU is how to map the irregular data access pattern to the GPU architecture. Bell and Garland (2008, 2009) discussed the sparse matrix format for SpMV, including ELL (ELLPACK), COO (coordinate), DIA (diagonal format), and CSR. Experimental results show that the scalar-based CSR format is not suitable for SpMV due to its low bandwidth utilization caused by non-coalesced memory access patterns, whereas the vector-based CSR format can achieve a good performance on matrices with large row sizes due to the contiguous memory accesses. Based on this conclusion, a hybrid (HYB) format is proposed in Bell and Garland (2009). In the HYB format, the ELL format is used to store non-zero values in each row and the COO data structure is used to store the remaining entries. Monakov proposed a hybrid BCSR/BCOO format in Monakov and Avetisyan (2009), where a CSR-like format is used to store the blocks. The row coordinate is stored by sorting the blocks by rows and then storing the index of the first block in each row in a CSR-like format. Compared to the blocked CSR (BCSR) format, the hybrid format is more flexible. The performance of SpMV with different data formats varies with different data characteristics (Li et al. 2015), hence the best format has to be chosen according to the dataset. To address the problem, Benatia et al. (2016) proposed a machine-learning approach to select the best representation method for a given spare matrix. Similar to Benatia et al. (2016), Su and Keutzer (2012) developed a SpMV framework, called clSpMV, where a *Cocktail* format is used to represent a sparse matrix. clSpMV analyzes SpMV at runtime and chooses the best representations of a given matrix. Although most of the previous studies are centered on memory access patterns and data representations, Yan et al. (2014) studied the load

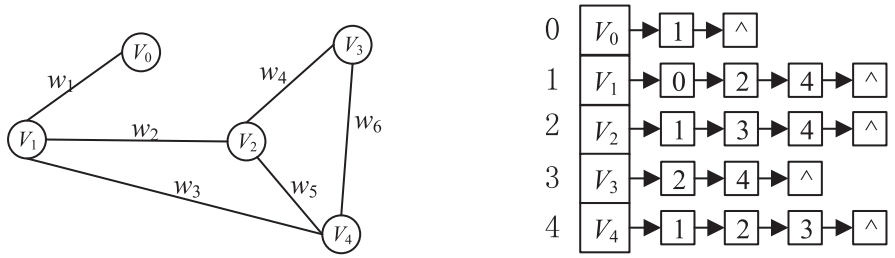
imbalance problem and developed the yaSpMV framework. yaSpMV addresses the load imbalance problem by revisiting the segmented scan approach for SpMV. By partitioning a matrix into strips of warp sizes, Zheng (BiE 2014) proposed the BiELL format to maintain load balance for SpMV.

Graph Partition. A graph partition algorithm cuts the vertices into several disjoint subsets, which is widely used in distributed large-scale graph processing and many other application scenarios, such as scientific computing, computer vision, and distributed job scheduling. Vineet and Narayanan (2008) implemented the push-relabel max-flow/min-cut algorithm on GPUs. The authors stored the vertices status information in the shared memory. Experiments on 640×480 images for 90 graph cuts gain 10–12× speedup over the best sequential algorithm reported in 2008. Recently, some researchers proposed the two-way cut algorithm. However, this method does not solve the problem of partitioning the graph into multiple sub-graphs. This is a problem called the minimum k-cut problem. The aim of the minimum k-cut problem is to partition the graph into k independent sub-graphs while every sub-graph is a connected one. When k is a part of the input, the minimum k-cut problem is NP-hard. The complexity is $O(|V|^{k^2})$ even with a fixed k. The main goal of graph partitioning is to achieve load balancing and facilitate task scheduling for static graphs. As the graph topology is static, the algorithm only needs to run once. The method is suitable for both CPU and GPU. For dynamic graphs in which the topologies change, it is difficult to implement such algorithm. Frog (Shi et al. 2015) partitioned the graph by using a hybird coloring model. The coloring algorithm in Frog is incomplete, which does not restrain all adjacent vertices from being labeled by different colors. Instead, the color number is set by the user while Frog only ensures the adjacent vertices are not colored by the small set of colors. For the rest of the vertices, Frog combined the vertices together into a single color and all the vertices in the same color are processed in a super-step. By using this method, if the graph is divided into N partitions, the color for the first $N - 1$ partitions is different and there is an edge between any pair of vertices in each partition. Therefore, the first $N - 1$ partitions can be processed in parallel. CuSha (Khorasani et al. 2014) first splits the vertices into P shards and the edges in a shard are listed based on the increasing order of their source vertices. By using this partition method, the edges of each vertex are stored in a continuous memory chunk, which can make the memory access regular.

MST. For an undirected graph, a minimum spanning tree is a connected subgraph, which connects all the vertices together with minimum total weight. Vineet et al. (2009) implemented the fast MST algorithm on CUDA by recursively calling the Boruvka algorithm. In their algorithm, they mapped the irregular steps of super-vertex formation and recursive graph construction to primitives such as split to categories involving vertex IDs and edge weights. In the proposed algorithm, in the first phase each vertex finds the edge with the minimal weight to the neighbor vertex. In the second phase, vertices are merged into disjoint components called vertices. The algorithm performs these two phases recursively, until there is only one super-vertex. In each iteration, the authors reorder the edges, put the edges with the vertices in a continuous memory chunk, and then remove the duplicate edges. By doing so, the memory access can be regular. Experiments on a NVIDIA Tesla S1070 show this method can achieve 8–10× and 30–50× speedup over their previous implementation and the serial implementation, respectively.

4 GPU GRAPH-PROCESSING FRAMEWORKS

Besides optimizing individual graph-processing algorithms, many researchers have also investigated how to build a general graph-processing system on GPUs. In this section, we survey the existing GPU graph-processing systems, including their data layout, parallel graph programming models, and their system implementations and optimizations.



(a) Example graph (b) Adjacency list representation of the graph.

Fig. 4. The adjacency list representation example of graph.

4.1 Data Layout Models

As the considerations of data layout are very similar in different graph algorithms and frameworks, we discuss the data layout in a single section. As for other GPU aspects, such as memory access pattern, workload mapping, and GPU-specific programming, we discuss them by referring to different types of graph algorithms. As mentioned earlier, the main requirements of data layout are the compactness and the regularity. The former minimizes the PCIe bandwidth consumption, while the latter enables the regular memory access and maximizes parallelism. We survey the main graph representations that are used in the existing GPU-based graph-processing algorithms and systems.

4.1.1 Adjacency Matrix and Adjacency List. The adjacency matrix and adjacency list are two basic graph representations, which have been widely used in early parallel graph-processing studies (Harish and Narayanan 2007; Narayanan et al. 2010; Merrill et al. 2012; Fagginger Auer and Bisseling 2012).

The adjacency matrix is a square matrix. In an unweighted graph, a non-zero element a_{ij} indicates there is an edge between the i-th vertex and the j-th vertex, while in a weighted graph, a non-zero element stands for the weight of the edge. For most large-scale graphs, the matrix is typically sparse. Some researchers directly use the existing libraries to handle the sparse matrix, such as CuSparse.[4] Katz and Kider (2008) use 2D texture to represent the adjacency matrix in GPU memory. The adjacency matrix representation simplifies the memory allocation for programmers. However, due to the sparsity of the adjacency matrix, the memory space is wasted.

Another typical graph representation is the adjacency list, which is a collection of unordered lists, each representing the set of neighbors of a vertex in the graph. Figure 4 shows an example of the adjacency list of an undirected graph. The adjacency list is more compact than the adjacency matrix. However, it does not enable regular memory access, because the neighbors of different vertices are not stored in a contiguous memory space. As discussed before, this may incur much more memory access when reading the data needed by the threads in a warp.

4.1.2 Vector Graph (V-Graph). V-graph is another efficient graph representation method proposed by Blelloch (1990). Figure 5 shows the v-graph representations of the same graph in Figure 4. In v-graph, the topology of an undirected graph is stored in a segmented vector (i.e., the Cross-pointer in Figure 5), where each segment corresponds to a vertex. Each element of a segment stores the *Cross-pointer* of an edge incident to the corresponding vertex. For example, in Figure 4, the element with index 3 in *Cross-pointers* has the value 9, which indicates that this edge is connected to the same vertex as the edge with index 9. For a directed graph, two segmented vectors are used: one storing the incoming edges of the vertices, while the other stores the outgoing edges. Additional

Index	= [0	1	2	3	4	5	6	7	8	9	10	11]
Vertex	= [1	2			3			4		5]		
Segment-descriptor	= [1	3			3			2		3]		
Cross-pointers	= [1	0	4	9	2	7	10	5	11	3	6	8]
Weights	= [w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6]

Fig. 5. The example v-graph representation of graph in Figure 4.

Row offset	0	1	4	7	9	11						
Column index	1	0	2	4	1	3	4	2	4	1	2	3
Values	w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6

Fig. 6. The example CSR representation of graph in Figure 4.

vectors are used to store other information, including the vertex degree (*Segment-descriptor*), the edge weights (*Weights*), and so on. In v-graph, all the edges are stored in a contiguous memory space sorted by their incident vertices, which therefore supports regular memory accesses and efficient GPU computations. However, v-graph is not a very compact graph representation, because it contains a lot of redundant information.

4.1.3 CSR. In order to achieve both compact storage and regular memory access, some graph algorithms, such as Merrill et al. (2012), make use of the CSR format. Figure 6 is an example of the CSR representations of the graph in Figure 4. In CSR, three one-dimensional arrays are used, each storing the non-zero values in an adjacency matrix, the offsets of the rows in the values array, and the column indices of the values, respectively. Just like the v-graph, CSR sorts and stores the information of all the edges of a vertex compactly in a contiguous chunk of memory one after another, which enables regular memory accesses, lowers the memory requirement, and reduces the PCIe bandwidth consumption when transferring data between the host and the GPU device.

4.2 Graph Programming Models

A general graph-processing system typically exposes a programming framework to the programmers, which consists of two components: a programming interface and a parallel programming model. The programming interface defines a set of APIs to facilitate the formulation of a graph computation. Existing graph systems on GPUs typically adopt a vertex-centric model, where programmers need to define a few functions that are executed on each individual vertex. The parallel programming model is an abstraction of the parallel computation architecture, which usually states how the parallel processes are formulated, and more importantly how they interact with each other, including communication and synchronization. Parallel programming model is a well studied area. A lot of models have been proposed in the literature. For example, there are a series of traditional parallel programming models such as the Actor model (Agha 1986), the Bulk Synchronous Parallel (BSP) model (Valiant 1990), the LogP machine model (Culler et al. 1993), the Dataflow model, and the Parallel Random Access Machine (PRAM) model (Asanovic et al. 2009).

Most existing GPU-based graph-processing systems provide a vertex-centric programming interface, with which the graph program is expressed in the functions that will be applied on

each vertex iteratively. Furthermore, two major parallel graph programming models are proposed, namely, Gather-Apply-Scatter (GAS) and BSP (Bulk Synchronous Parallel).

4.2.1 GAS Model. GAS is a popular parallel graph programming model used in a lot of graph-processing systems (Gonzalez et al. 2012). Several existing GPU-based graph-processing systems adopt the GAS model, such as VertexAPI2 (Elsen and Vaidyanathan 2013), MapGraph (Fu et al. 2014), and CuSha (Khorasani et al. 2014). These systems typically provide a vertex-centric programming interface that contains three major functions: Gather, Apply, and Scatter. In the GAS model, the program on each vertex can be divided into three phases, which are listed as follows.

—The gather phase. In this phase, a vertex collects the information from the adjacent vertices and edges by using the user-defined gather function.
—The apply phase. The user-defined apply function is called on a vertex based on the information collected in the gather phase. The vertex's value(s) is updated in this phase by calling the apply function. This is the only phase without communications between vertices.
—The scatter phase. In this phase, the new value(s) of the vertex is scattered to its adjacent vertices and edges. In some implementations, the push-style scatter is used, which pushes the updates to remote vertices. With the push-style scattering, some traversal algorithms can simply disregard the gather phase so that the edge traversals can be reduced.

The GAS model abstracts away the synchronization overhead, which simplifies the analysis process for the complexity and the correctness of the graph algorithms implemented using this model. However, as the synchronization overhead in GPUs is not negligible, we cannot ignore it when we implement a graph-processing system that supports this model. For instance, CUDA only supports the synchronization among threads in the same block. To achieve a global synchronization among all the threads in different blocks, the system can split the computation into a number of kernels. Since the GPU executes the kernels one after another, the end point of each kernel effectively acts as a global barrier. Both a local block-wise synchronization and a global synchronization using a number of kernels are very expensive. Therefore, a critical challenge in the implementation of a graph-processing system is to minimize the number of synchronization points.

4.2.2 BSP Model. A program in a BSP model (Valiant 1990) is executed in a sequence of so-called super-steps. Within each super-step, the parallel processes run asynchronously and communicate with each other by sending and receiving messages. At the end of each super-step, all the processes are synchronized by using a barrier. This procedure is shown as Figure 7. More specifically, a super-step in each process consists of the following three phases:

—Local computation: the computation tasks are executed locally.
—Global communication: all the communications, including sending and receiving messages, are executed in this phase.
—Barrier synchronization: all the computation and communications are synchronized and guaranteed to be completed at this point.

Pregel (Malewicz et al. 2010) is probably the first graph-processing system that adopts the BSP model to implement a vertex-centric parallel graph programming interface. In this model, within each super-step, a user-defined function is applied on each vertex asynchronously, which updates the value on the vertex, and the updated values of the vertices are then sent to their neighbors by passing messages. One iteration of the function executions and message passing on all the vertices will be completed and synchronized at the end of a super-step. In addition, the vertex will be executed only when it receives a message in subsequent super-steps.

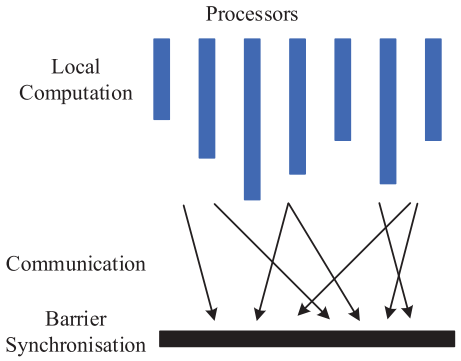


Fig. 7. The BSP Computing Model.

Representative GPU-based graph-processing systems that use the BSP model include TOTEM (Gharaibeh et al. 2012), Medusa (Zhong and He 2014), and GunRock (Wang et al. 2016). In these systems, a large graph is usually divided into several partitions and one user-defined kernel will be run on each graph partition. In a super-step, all the threads in the kernel are run concurrently. Within a kernel, each thread receives the messages from the previous super-step and then performs the local computation. In the local computation phase, the values of the vertices are stored at the local memory to minimize data transfer. Each kernel can send the messages to its neighbors if necessary before the end of the current super-step. A barrier is imposed between two super-steps to synchronize all the kernels. A main disadvantage of the BSP model is that it may suffer from the straggler problem, where the thread with the longest execution time can delay all the other threads in a super-step.

4.3 Data Layout

Due to the mismatch between the irregularity of graph-processing algorithms and the symmetric hardware architecture of GPU, applying traditional graph-processing methods on GPU will inherently suffer from the problem of underutilizing the GPU's capability. Some high-performance graph-processing systems attempt to solve these problems through designing a compact storage and regular memory access data layout. For example, TOTEM (Gharaibeh et al. 2012), Medusa (Zhong and He 2014), MapGraph (Fu et al. 2014), and Frog (Shi et al. 2015) use the CSR format to represent the graph structure. As discussed in Section 4.1, CSR is a regular graph representation, but accessing the neighbors of a vertex will lead to poor locality, which causes lots of random input-dependent memory accessing (also known as *non-coalesced memory accesses*). In addition, CSR is hard for some update operations such as adding or deleting a vertex. In order to overcome the *non-coalesced memory accesses* problem with CSR, CuSha (Khorasani et al. 2014) implemented the *shard* technique (Kyrola et al. 2012) on GPU, which is widely used in disk-based graph-processing systems such as GraphChi (Kyrola et al. 2012) and VENUS (Cheng et al. 2015). In CuSha, the GPU implementation of shard was called as *G-Shard*. The *shard* technique first sorts the vertices in an ascending order and partitions them into equal-sized *windows*. For each window, a shard is created to store all the edges connected to the vertices in the window. Furthermore, all the edges in a shard are sorted according to the IDs of their source vertices. In this way, the graph data in each shard is organized according to the accessing order of the vertices. *G-Shard* adopts the same way as *shard* to organize the vertices and the edges. In addition, G-Shard revised the *window* as the *Concatenated Window* (CW), which lists the edges related to the *window*, so that each thread can

visit the vertices according to the CW list. In CuSha, each G-Shard corresponds to a thread block. G-Shards can lead to a better locality, as all the vertices are continuous and all the edges of a vertex are stored in a continuous chunk. On the other hand, G-Shards are disjoint with each other; the computation on different G-Shards can be performed asynchronously, which is well matched with GPU. By using the G-Shard, the graph data is well organized in CuSha, which enables the memory access to be coalesced.

As G-Shard equally partitions the vertices, the CW size differs as the vertex degree differs. Therefore, it is easy for the G-Share technique to encounter the warp divergence problem. In order to solve this problem and update the graph data efficiently, GStream (Seo et al. 2015) and GTS (Kim et al. 2016) use the slotted page format proposed by TurboGraph (Han et al. 2013) to store the graph in disk and memory. In the slotted page representation, the graph is partitioned into a list of slotted pages, with the size of each page being several MBytes. The vertices ID and its adjacency lists are stored in a slotted page consecutively. In most cases, since the adjacency list of a vertex is smaller than the size of a single page, multiple adjacency lists can be stored in one page, which is called *Small Adjacency* list page (SA page). In the power-law graph, there also exist some vertices with the sizes of their adjacency lists bigger than one page. Then, several pages are needed to store the adjacency list of the vertex. Consequently, one of those pages stores the information regarding only one adjacency list. This type of page is called *Large Adjacency* list pages (LA pages). This representation is not very compact compared with CSR, but makes it much easier to update the graph data. Compared with G-Shard, it is much easier for this method to allocate the memory space as the page size is fixed, while the window size of G-shard changes.

4.4 Memory Access Pattern

GraphReduce (Sengupta et al. 2015) is a CUDA/C++ library for large-scale graph processing. GraphReduce presents a set of APIs, aiming to hide the GPU programming details. In order to process large-scale graphs which cannot be loaded into memory, GraphReduce partitions the graphs into small sub-graphs with approximate sizes, and sorts the edges in the sub-graph according to the source's vertex to match the memory access patten in GPU. Aiming to leverage GPU memory coalescing and pre-fetch the unvisited data into memory for the sequential accesses, GraphReduce adopts the Unified Virtual Addressing (UVA) to allocate the memory space and uses the DMA technology to directly translate the memory loading/storing operations over the PCIe. By using these methods, the memory accesses are sequential and the communications can be overlapped with GPU computations through pre-fetching.

In order to maintain the regular memory access, GStream (Seo et al. 2015) introduced the concept of the "join" operation from the database area and proposed a "nested-loop theta-join" operation, which achieved the coalesced memory access by parallelizing the read-only and the read/write operations in parallel. In the nested-loop theta-join method, the vectors of the read/write and the read-only attributes are denoted by WA and RA, respectively, and the topology data by SP. GStream divided the WA into W partitions because the values change frequently during the computation phase. Since WA is updated frequently during an iteration of graph processing, GStream stores the WA_i data in the device memory to improve the system performance. While the RA and SP are the constant data, GStream fed the RA data and the corresponding SP data into the device memory. In GStream, an asynchronous data transfer technique such as the overlapping technique in GraphReduce was used to improve GPU utilization by hiding the memory access latency. Unlike other GPU graph-processing systems, GStream is a pure GPU graph-processing system, all the computation tasks were finished on the GPU processor, and the CPUs were not involved in the computation phase.

GTS (Kim et al. 2016) processes the entire graph only using GPUs. In order to overcome the limited memory capacity in the GPU device and even in the host, GTS uses the CUDA streaming method to transfer the unvisited graph data to the GPU device memory and swap the visited data to the disk. By using this method, there is no need to partition the graph. Namely, GTS can process large-scale graphs without pre-processing. GTS distinguishes the graph data by tagging *attribute data* and *topology data*. The attribute data refers to the information of the vertices and the edges (e.g., the weight of the edge and the value of the vertex) that are required and updated during the execution of the vertex kernels, while the topology data is the basic structure data of the graph. GTS stores the graph in PCIe SSDs and triggers the direct data transfer by GPU. Thousands of GPU cores can be used when streaming the topology data from SSDs to GPUs through PCIe. To be more specific, the attribute data was copied into the GPU device memory, and then the topology data was copied from the host memory to the GPU device memory in the streaming method, where the data was processed by the user-defined GPU kernel function. GTS adopts a similar method as GStream to store the attribute data in the device memory and swaps the topology data to the disk. In GTS, by using the asynchronous GPU streaming method (e.g., CUDA Streams), the data can be transferred asynchronously, which can overlap the latency of the memory access from GPUs to main memory and can also improve the GPU utilization. Compared with GraphReduce, the pre-processing phase can be removed in GTS.

4.5 Workload Mapping

As mentioned before, the workload in graph processing is irregular because of the variance in the vertex degree. How to map the uneven workload of each vertex onto the GPU greatly affects the processing efficiency of GPU. One of the most important existing works in this area balances the workload by cooperation among threads. The *dynamic scheduling* and the *two-phase decomposition* strategies are used in MapGraph (Fu et al. 2014) to gain better performance in workload mapping. The *Dynamic scheduling* strategy combines three scheduling strategies, i.e., *CTA-based, scan-based*, and *warp-based*, to achieve higher performance in workload mapping (here CTA is short for Cooperative Thread Array). The CTA-based scheduling strategy distributes the workload to the threads in a CTA according to the vertex degree. In this strategy, the workload of the vertices in the frontier is assigned to the whole CTA and every thread in the CTA serves only one vertex. The number of threads in a CTA is much more than that in a warp, which made the CTA-based scheduling strategy suitable for the vertices with large degrees. MapGraph uses a different scheduling strategy according to the vertex degree. MapGraph first applies the CTA-based scheduling strategy to the vertices with adjacency lists larger than the CTA size. Next, it performs the warp-based scheduling strategy for the vertices with the degrees larger than the warp width but smaller than the CTA size. Finally, it applies the scan-based scattering strategy for the "loose ends" vertices whose degrees are smaller than the warp width. Although *dynamic scheduling* achieves relatively good performance for the many-graph algorithm such as SSSP and BFS, there are still some drawbacks with this method.

On one hand, because of these three separated stages of this strategy, the parallelism among the stages is lost and hence the degree of parallelism of the instructions decreases. On the other hand, as each thread in the scan part of the graph algorithm needs to communicate with the thread processing its neighbor vertices in the CTA, other threads have to wait until all threads in a CTA are loaded. Finally, by using this strategy, the equal number of frontier vertices are assigned to a CTA, and therefore the total number of the handled adjacent vertices may be much more than the CTAs. This will lead to the imbalanced workloads among CTAs.

In order to solve the uneven workload mapping problem of *dynamic scheduling*, MapGraph proposes a *two-phase decomposition* scheduling strategy. This strategy attempts to achieve the

optimal workload mapping performance for threads within and across CTAs. The fundamental idea of this strategy is to decompose the scattering process into two phases: the scheduling phase and the computation phase. Unlike the dynamic scheduling strategy, the two-phase decomposition scheduling strategy is used to assign the adjacent edges to a CTA, which ensures the number of edges is same as the CTA size. In this strategy, the target of assigning the adjacent edges is achieved by finding the intersection between the starting and the ending points of each CTA, which are within the column-indexed array by using the sorted method. In the communication phase, the same number of adjacent vertices are visited by each thread. This scheduling strategy solves the problem of uneven workload mapping in dynamic scheduling strategy. But the overhead is relatively high.

GunRock (Wang et al. 2016) integrates the technologies proposed by Merrill et al. (2012) and Davidson et al. (2014). Then the author proposes two workload mapping strategies, which are called *per-thread fine-grained* and *per-warp & per-CTA coarse-grained*. In the *per-thread fine-grained* strategy, one thread maps to the neighbor list of a frontier vertex. In this method, each thread loads the offset in the adjacency list of the assigned node. Next, all the edges in the adjacency list are processed sequentially by the thread. Considering the significant difference in the workload performance with the per-thread fine-grained strategy, which is caused by different adjacency list sizes, GunRock proposes a *per-warp & per-CTA coarse-grained* strategy. In this strategy, the workload mapping problem is solved by dividing the adjacency list into three categories according to the size of the adjacency list and then mapping each category to a strategy which targets specifically at the corresponding size. These two strategies focus on different task granularities. The experiments show that the *per-thread fine-grained* strategy works better with the graph with a large diameter and the relatively even degree distribution. This strategy balances the threads well in the CTA, but does not work well across CTAs. On the contrary, the *per-warp & per-CTA coarse-grained* strategy performs better for the power-law graph, which has an uneven degree distribution.

As part of the physical warp, a virtual warp controls the tradeoff between GPU utilization and path divergence. Generally, 2, 4, 8, 16, or more virtual warps constitute a physical warp. So the processing task can be performed iteratively because the iteration is performed separately by different GPU kernel calls. In a virtual warp, several threads process a vertex concurrently and each thread in the virtual warp works in parallel. Using this method, the read and computation phase are finished by different threads in the virtual warp. Compared with the workload mapping strategy in GunRock, the virtual warp is a more general method with equally sized virtual warp, while the strategy of GunRock is more flexible for different task granularities.

4.6 Miscellaneous

GraphReduce adopts the Gather-Apply-Scatter programming model. In real-world graphs, the number of edges is much more than the number of vertices. In the gather and scatter stage, the message passing about the edges is much more than that about the vertices. In the apply stage, the computation for the vertices is much less than that for the edges. In order to reduce the communication cost and improve the parallelism, both vertex- and edge-centric programming methods are used in GraphReduce. The authors use the edge-centric programming method in the gather and scatter stage, and use the vertex-centric programming in the apply stage to improve the parallelism.

Medusa (Zhong and He 2014) and MapGraph (Fu et al. 2014) both provided a set of APIs for graph processing on GPUs. By using the APIs provided by Medusa, the programmers can define their own functions for processing vertices, edges, and messages. In order to improve the programmability and usability, Medusa encapsulates the frequently used system operations to overlap the GPU-specific programming details. In addition, in order to enhance the flexibility, Medusa

provides a set of configuration parameters and utility functions to control the iteration executions. MapGraph (Fu et al. 2014) is a high-performance parallel graph programming framework, which also provides a set of flexible APIs with high programmability based on GAS. In MapGraph, programmers can define the computation functions on vertices and edges by invoking the MapGraph kernels. MapGraph uses the same method as Medusa to enhance the flexibility by providing a set of configuration parameters. In addition, a set of utility functions are provided by the library calls and used for the iteration control and other functionalities. GunRock (Wang et al. 2016) offers an easy-to-use programming interface by implementing a data-centric abstraction. Unlike other vertex-centric and edge-centric programming methods, GunRock's data-centric abstraction focuses on the operations of the frontier of vertices or edges, which makes the programming interface easy to use.

Frog (Shi et al. 2015) is a graph-processing framework which has the lightweight asynchronous scheme. A hybrid-coloring model is proposed for graph partition and a streaming execution engine is designed for asynchronous processing in Frog. As graph coloring is a complex algorithm, an incomplete coloring scheme and the Pareto principle are used as a compromise. In the graph coloring algorithm, the vertices with the same color are disjoint. Therefore, all the vertices and edges in the same color (partition) can be processed in parallel. As the incomplete coloring scheme is used, Frog divides the first $n - 1$ coloring steps into the P-step and the last coloring step into the S-step, according to the aforementioned analysis. All the vertices and edges in the P-step can be processed in parallel, while the S-step is handled sequentially by atomic operations. Benefiting from the asynchronous processing method in Frog, the transferring of the data can be overlapped with the execution of the kernel function, which improves the system performance.

Modern GPUs can offer a very high degree of parallelism when the graph processing is regular. It is a great challenge to effectively exploit the parallelism potential of GPU. In addition, the GPU memory is limited compared with the ever-increasing graph size. Hence, we need to copy the data into and out of GPU during the graph processing. It is another critical issue to design an efficient communication method between CPU and GPU. Designing a suitable data layout can help tackle the above two issues. On one hand, with a smart data layout, the graph-processing algorithm can match the graph data to the memory architecture of GPU and enable the regular memory access. On the other hand, a well designed data layout can reduce the communication cost. Two widely used techniques of speeding up the memory access are (i) coalescing the memory access requests from a set of parallel threads, and (ii) prefetching the unvisited data to the memory to overlap communication with computation.

5 EXPERIMENTS

In this survey, we implemented a few commonly used graph algorithms and conducted experiments with these algorithms and a number of graph-processing frameworks. On one hand, we compare the performance of the graph processing systems with different types of graphs, such as graphs following the power-law and graphs with large diameters. On the other hand, we compare the performance of different graph processing algorithms when they are implemented with GPU- or CPU-based graph processing frameworks, respectively. This verifies the benefit of using GPU for graph processing.

5.1 Experimental Configurations

5.1.1 Experimental Datasets. The real-world graphs have different characteristics. In this section, we mainly focus on the typical graph datasets with comparable data format. In this article, all the datasets are represented in the classic graph formalism method (West 2001). V represents the collection of verities, E is the set of edges which connect the vertices, and $G = (V, E)$ represents

Table 2. Experimental Datasets

Datasets	Amazon	DBLP	RoadNet-CA	WikiTalk	Twitter	YouTube
Vertices	735,322	986,286	1,965,206	5,533,214	41,652,229	1,134,890
Edges	5,158,012	6,707,236	5,533,214	5,021,410	1,468,365,167	2,987,624
Directivity	directed	undirected	undirected	directed	directed	undirected

the graph. There is an edge between vertex u and v only when the two vertices are connected. In addition, the edge is presented as $e = (u, v)$ or $e = <u, v>$ for undirected or directed graphs, respectively. Both directed and undirected graphs are considered in this article.

Considering the characteristics of power-law and large diameter in real-world graphs, we select six graphs with different structures and a varying number of vertices and edges. All six graphs are shown in Table 2. All the graphs are stored in a plain text file and all the graph data are organized by a processing-friendly format without indices. In the file, the integers are used to identify the vertices, with a line storing one vertex. For the undirected graphs, the vertex ID and the adjacency list are included, while for the directed graph, the vertex ID and two adjacency lists, which correspond to the incoming and outgoing edges, are included in a vertex line.

The number of vertices and edges in the selected graph datasets are shown in Table 2. The graphs are selected from diverse sources, including e-business, social network, citation link, and other sources of real-world graphs with different sizes and graph metrics. The degree of the graphs ranges from 2 to 1,663. The graphs are extracted from the real-word problems, which have been shared in the Stanford Network Analysis Project (SNAP) (Leskovec 2009).

5.1.2 Experimental Algorithms. The graph processing algorithms we implemented include PR, BFS, SSSP, and CC. We selected these algorithms because they have different characteristics and can be used to test different aspects of performance.

PR uses the edge consistency model. When the rank value of vertex v is updated, the rank values of all neighboring vertices that have outbound edges are also updated. If the algorithm is implemented based on BSP, the rank values changed in the current iteration can only be observed by other vertices in the next super-step.

BFS is a commonly used graph traversal algorithm. The computation in BFS is very limited, while the communication is rather intensive. Due to this feature, a large number of memory lookup operations are used. Hence, the performance is related with the memory access pattern. Medusa is based on the kernel implementation of BFS, which explores all neighboring vertices in a level-by-level fashion from the first vertex.

SSSP tries to find the shortest path from a given vertex to other vertices in the graph. Dijkstra's algorithm is the traditional method to solve the SSSP problem.

CC is an algorithm extracting the subgraphs in which all the vertices are connected and there are no additional vertices.

There is the textbook implementation for BFS. As for CC, PR, and SSSP, there are different implementations. According to the reported performance of these implementations, we use Dijkstra's algorithm to implement the SSSP algorithm, which is a cloud-based connected component algorithm created by Wu and Du (2010). The implementations of BFS and PagRank are presented in the relevant experiments.

5.1.3 Graph-processing Frameworks. We select seven popular graph-processing frameworks, namely, TOTEM, Medusa, GunRock, Frog, and GraphChi, and compare their performance in our experiments. The first four systems are GPU-based while GraphChi is CPU-based.

Medusa is an optimized graph-processing system with a set of simplified programming interfaces. Since Medusa requires loading the entire graph into the GPU device memory all at once, only the graphs whose sizes are smaller than the device memory can be processed. TOTEM is a hybrid system, which partitions the graph into two parts, one being processed by GPU while the other is processed by CPU. There are three partition strategies in TOTEM: HIGH-degree, LOW-degree, and RAND-degree partitions. In the HIGH-degree partition strategy, the vertices with the highest degree are assigned to CPU, while the LOW-degree vertices are assigned to GPU. LOW-degree is opposite to the HIGH-degree strategy. The RAND-degree strategy assigns the vertices to CPU and GPU randomly or sets the percentage of the edges that are assigned to different devices. In our experiments, in order to make full use of the computing power of GPU, we load all graph edges onto the GPU device. Unlike Medusa and TOTEM, CuSha is a vertex-centric graph-processing framework, which uses the new graph representations known as Concatenated Windows (CW) and G-Shards. GunRock is a high-performance graph-processing library for GPUs. The input parameters in our evaluation are the same as those used in the corresponding publications.

5.1.4 Hardwares. We conducted the experiments on a Tesla-based GPU (NVIDIA Tesla K20m with 5GB device memory and 2,496 CUDA cores). The programs are written with CUDA 7.5 using the "-arch=sm 35" flag. We ran GraphChi (Kyrola et al. 2012) on a machine with 8GB memory and two Intel(R) Xeon(R) E5-2670 CPUs, each at 2.60GHz. We reused the source code of these graph-processing engines given by the authors directly. The experiments were all conducted on RedHat 4.4.5-6.

5.2 Experiment Results

In order to identify the types of dataset and algorithms that can be processed efficiently on GPU, we first conduct the experiments and compare the runtime of different algorithms with different datasets. The experimental results are shown in Table 3. Table 3 shows that, even though the graph has an irregular structure and GPU performs the best with regular data access, the tested algorithms achieve better performance on GPU than on CPU. The difference in performance between Wiki-Talk and Twitter indicates that the situation becomes worse as the data size increases. In PageRank, the updated vertices need to send their values to the neighboring vertices before the next iteration begins. Therefore, the communication cost plays an important role in the performance of PageRank. As we have discussed in Section 2.2, the GPU device is connected to the host through the PCIe bus. But the PCIe bandwidth is limited. This is the reason why the situation deteriorates when the data size becomes bigger than the GPU memory. In comparison, Amazon, DBLP, Wiki-Talk, YouTube, and Twitter are power-law graphs, while RoadNet-CA is a graph with a large diameter and almost the same degree for each vertex. The performance between RoadNet-CA and the datasets indicate that the acceleration effect with sparse graph is not as high as with power-law graphs.

In order to investigate the types of data layout, memory access pattern, workload mapping, and some other factors such as the branch divergence, we measure the memory throughput, the active warp in every SM cycle, load efficiency of the global memory, memory copy time, and the bank-conflict of the systems. The results are listed in Tables 4–9. We analyze the results in these tables from the following four perspectives.

5.2.1 Data Layout. We measured the ratio of the requested global memory throughput to acquired global memory throughput (also called *gst_efficiency*) of each system. Values greater than 100% indicate that, on average, multiple threads in a warp access the same memory address. In other words, the *gst_efficiency* indicates whether the data is aligned or not. The result is listed in Table 4. From this table, we can see that GunRock and CuSha have higher *gst_efficiency* on all the

Table 3. Execution Time (in Milliseconds)

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	26.91	42.6	425.51	34.9	NULL	42.23
	Frog	11.68	7.64	232.99	4.86	NULL	5.7
	CuSha	27.914	19.66	374.956	2.984	NULL	0.254
	Medusa	19.099	7.051	201.357	4.698	NULL	5.276
	GunRock	5.6936	3.792	8.2052	6.244	9.2232	8.324
	GraphChi	606.797	939.039	909.198	3362.79	658.124	NULL
PR	TOTEM	21.86	34.06	56.39	70.82	NULL	44.99
	Frog	35.52	49.61	22.78	38.13	NULL	17.91
	CuSha	28.013	19.748	374.873	2.987	NULL	0.253
	Medusa	278.587	567.612	317.532	8,757.567	NULL	2,739.184
	GunRock	62.6958	60.57	49.536	55.8162	62.244	58.086
	GraphChi	557.177	719.923	963.803	1,018.5	640.803	519.538
SSSP	TOTEM	41.82	27.43	821.39	57.55	NULL	13.12
	Frog	16.45	12.63	50.08	12.2	NULL	11.16
	CuSha	27.923	19.768	375.811	2.971	NULL	0.3
	Medusa	3.276	4.713	1.881	57.273	NULL	18.411
	GunRock	15.6443	13.3024	7.3367	10.904	13.8771	11.6144
	GraphChi	475.702	634.708	641.975	858.323	NULL	455.166
CC	TOTEM	48.3	45.07	1421.82	99.39	NULL	25.7
	Frog	8.88	9.47	103.54	11.3	NULL	7.51
	CuSha	27.905	19.7	375.988	2.991	NULL	0.252
	GunRock	23.403883	23.454189	25.617838	23.898125	25.456905	23.092031
	GraphChi	1,490.21	1,905.63	1,923.72	2,338.4	NULL	1403.35

Note: Null means that the system cannot process such dataset.

datasets and algorithms than any other system, which indicates that GunRock and CuSha have better organization of the graph data (the experiments in Section 5.2.2 can also draw a similar conclusion).

5.2.2 Memory Access Pattern. In order to investigate the memory access patterns of the systems, we first measured the ratio of the memory copy time to the whole execution time as shown in Table 5. Both the data copying from the host to the device and from the device to the host are measured in our experiment.

Table 5 shows Medusa has a higher host-to-device memory copying ratio than any other system with BFS and SSSP on Amazon, DBLP, and RoadNet-CA. We can also conclude that Frog has higher device-to-host memory copying than other systems, except SSSP and CC with RoadNet-CA on TOTEM and SSSP with YouTube on Medusa, from this table. This is because, in the computation phase, the communication of Frog is overlapped with GPU computation. But when the computation is completed, Frog needs to transfer the computation result to the host for combination. This is why Frog has the highest device-to-host memory copying ratio. By using the Edge-Message-Vertex (EMV) model, Medusa decouples the single vertex API into several separate APIs, which improves the processing efficiency, but on the other hand it also leads to more memory copying operations. By using the G-shard technology in CuSha, the vertices are sorted in every shard and the shard can be disconnected. By using this technology, the communication can be completely overlapped with the computation phase when CuSha sends the result back to the host. This is why CuSha has the lowest host-to-device memory copying ratio than other systems.

Table 4. Ratio of Requested Global Memory Store Throughput to Required Global Memory Store Throughput (%)

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	62.021	51.1614	55.3328	47.18	NULL	44
	Frog	16.3775	16.3175	15.5975	9.9425	NULL	14.33
	CuSha	73.35	71.53	80.74	41.9	NULL	0
	Medusa	14.17	14.57	10.6	11.28	NULL	14.46
	GunRock	69.75	71.34875	68.51	69.72	69.35	68.735
PR	TOTEM	69.375	64.5825	71.25	71.415	NULL	70.09
	Frog	16.3775	16.3175	15.5975	9.9425	NULL	14.33
	CuSha	73.33	71.13	80.74	41.9	NULL	0
	Medusa	71.797	71.283	71.39	70.87	NULL	70.86
	GunRock	90.61	90.54	90.58	90.56	90.48	90.475
SSSP	TOTEM	39.7575	30.8925	40.7525	29.5575	NULL	27.82
	Frog	16.3775	16.3175	15.5975	9.9425	NULL	14.33
	CuSha	73.35	71.12	80.75	41.9	NULL	0
	Medusa	66.95	66.49	67.31	66.687	NULL	64.71
	GunRock	79.97	80.05	79.46	77.26	77.86	74.83
CC	TOTEM	63.65	59.84	57.81	58.687	NULL	58.24
	Frog	16.3775	16.3175	15.5975	9.9425	NULL	14.33
	CuSha	73.35	71.12	80.75	41.9	NULL	0
	GunRock	73.57	65.88	73.93	74.27	74.21	74.25

Note: Null means that the system cannot process such dataset.

The global memory throughput is shown in Table 6 and the ratio of active warps to the total warps in a single SM in shown in Table 7. Coalesced memory access has the highest impact on throughput, moreover, misaligned data format and non-coalesced memory access will lead to too much unnecessary load operations. Table 7 and Table 6 show that Frog and Medusa have higher throughput and active occupancy ratio than the other systems when running BFS and SSSP, which indicates that Frog and Medusa have better parallelism than other systems with BFS and SSSP. But Table 8 shows CuSha has the highest load efficiency. Note that Cusha uses the CSR format while Frog and Medusa use array data layout to achieve the coalesced memory access. So the result in Table 8 indicates that the CSR is better at enabling regular memory and reducing unnecessary load operations. Meanwhile, the frontier data layout is used in GunRock, which achieves better performance than other systems for BFS. This phenomenon indicates that, although data layout has a great effect on the memory access, it is not the only factor for system performance and there is no data layout that is superior with all algorithms.

Table 3 and Table 6 show GunRock and TOTEM have lower throughput than other systems, and they can also achieve better performance than the other systems on some algorithms, such as BFS and PageRank. GunRock can achieve the best performance on BFS. This is because the *enactor*, the core of GunRock kernel, combines multiple logical operations into one single kernel. By using this technique, GunRock can significantly save memory bandwidth. On the other hand, TOTEM achieves the best performance in PageRank when running on Amazon. TOTEM takes the communication rate into its performance model, and adopts data pre-fetching and caching methods to improve the efficiency of PCIe communication, which is important to the performance of PageRank.

Table 5. Memory Copy Time to The Whole Execution Time (%)

		Frog		TOTEM		CuSha		Medusa		GunRoack	
		HtoD	DtoH	HtoD	DtoH	HtoD	DtoH	HtoD	DtoH	HtoD	DtoH
Amazon	BFS	44.77	13.18	39.27	2.41	37.69	1.13	66.61	4.99	44.05	7.83
	PR	44.77	13.18	27.53	4.37	37.76	1.13	13.64	0.97	11.15	2.66
	SSSP	44.77	13.18	41.08	7.76	37.64	1.16	96.07	1.67	31.14	9.05
	CC	44.77	13.18	25.98	13.28	38.31	1.12	NULL	NULL	50.72	6.88
DBLP	BFS	47.62	13.96	52.27	2.24	52.85	1.47	67.48	6.32	43.45	7.71
	PR	47.62	13.96	24.24	5.56	52.17	1.48	5.45	0.63	11.35	2.67
	SSSP	47.62	13.96	55.18	4.31	52.8	1.46	91.62	7.11	29.44	7.46
	CC	47.62	13.96	26.52	8.62	52.19	1.49	NULL	NULL	47.43	7.33
RoadNet-CA	BFS	15.12	11.68	7.44	3.03	5.13	0.63	83.32	2.28	43.22	7.87
	PR	15.12	11.68	18.76	5.62	5.01	0.64	14.5	3.48	11.27	2.64
	SSSP	15.12	11.68	37.65	21.86	5	0.62	90.27	9.35	28.6	7.86
	CC	15.12	11.68	33.41	33.02	5	0.62	NULL	NULL	50.79	6.66
Wiki	BFS	40.85	34.78	36.2	5.18	80.07	6.62	73.14	17.68	43.59	7.87
	PR	40.85	34.78	15.06	6.93	80.08	6.62	0.65	0.2	11.3	2.68
	SSSP	40.85	34.78	39.87	6.1	79.96	6.74	77.24	8.78	29.26	8.01
	CC	40.85	34.78	15.3	5.46	80.09	6.61	NULL	NULL	50.44	6.79
Twitter	BFS	40.85	34.78	NULL	NULL	NULL	NULL	NULL	NULL	43.23	7.83
	PR	40.85	34.78	NULL	NULL	NULL	NULL	NULL	NULL	11.4	2.67
	SSSP	40.85	34.78	NULL	NULL	NULL	NULL	NULL	NULL	30.28	7.67
	CC	40.85	34.78	NULL	NULL	NULL	NULL	NULL	NULL	50.57	6.83
YouTube	BFS	40.85	34.78	49.45	7.35	92.03	1.67	83.15	7.34	43.25	8.78
	PR	40.85	34.78	14.14	5.83	92.2	1.65	0.67	18	11.4	2.74
	SSSP	40.85	34.78	69.23	9.17	92.21	1.64	74.06	74.06	30.4	7.63
	CC	40.85	34.78	37.57	16.48	91.79	1.68	NULL	NULL	51.22	6.86

Note: HtoD means the data copying from host to device and DtoH means the data copying from device to host. *Note*: NULL means that the system cannot process such dataset.

Table 8 shows the ratio of the used global load throughput to the system's global load throughput (which is also called the *gld_efficiency*); according to this table, we can see that the *gld_efficiency* of Frog and Medusa is not the highest, which explains why Frog and Medusa cannot achieve the best execution performance with their relatively high global memory throughput and active warp occupancy.

5.2.3 Workload Mapping. The ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor is shown in Table 7, and the average number of instructions executed by each warp is shown in Table 9. According to these two tables, we can find the number of instructions executed in a cycle. The larger number of instructions executed in a cycle indicates, on the one hand, a more efficient usage of the computing resource, and on the other hand, the higher risk of bank conflicts and warp divergence. Tables 7 and 9 show that GunRock has the lowest active warp occupancy and the most stable average number of instructions executed by each warp of GunRock, which means GunRock can achieve more stable performance than other systems. This is because GunRock adopts the *per-thread fine-grained* and *per-warp & per-CTA coarse-grained* workload mapping strategies, which can dynamically choose different workload mapping methods according to the task granularity.

Table 6. Global Memory Throughput (GB/S)

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	5.95	6.577	7.91	8.567	NULL	3.78
	Frog	122.01	122.01	110.1	72.0575	NULL	133.65
	CuSha	89.417	80.166	117.92	58.455	NULL	121.28
	Medusa	92.447	95.616	98.862	83.102	NULL	98.017
	GunRock	2.22	2.24	2.22	2.22	2.86	2.27
PR	TOTEM	66.96	68.94	70.026	70.109	NULL	60.69
	Frog	122.01	122.01	110.1	72.0575	NULL	133.65
	CuSha	89.356	80.069	118	58.816	NULL	121.02
	Medusa	81.6	72.67	78.83	74.55	NULL	72.74
	GunRock	1.74	1.74	1.75	1.75	1.74	1.74
SSSP	TOTEM	17.183	17.66	18.09	19.19	NULL	11.53
	Frog	122.01	122.01	110.1	72.0575	NULL	133.65
	CuSha	89.419	80.271	117.94	58.518	NULL	121.22
	Medusa	114.9	101.22	135.58	87.27	NULL	85.05
	GunRock	2.29	2.17	2.27	2.04	2.06	2.145
CC	TOTEM	44.06	46.75	49.59	52.94	NULL	39.32
	Frog	122.01	122.01	110.1	72.0575	NULL	133.65
	CuSha	89.346	80.22	117.95	58.382	NULL	120.86
	GunRock	6.73	6.59	6.72	6.82	6.75	6.76

Note: NULL means that the system cannot process such dataset.

Table 7. Active Warp Occupancy

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	0.463306	0.4228	0.4315	0.456212	NULL	0.3995
	Frog	0.8348	0.8364	0.7615	0.775	NULL	0.760239
	CuSha	0.933773	0.933685	0.976658	0.892485	NULL	0.978875
	Medusa	0.742502	0.742878	0.743046	0.736742	NULL	0.731812
	GunRock	0.337857	0.34443	0.337871	0.337814	0.336987	0.3389
PR	TOTEM		0.607594	0.605802	0.581388	NULL	0.5899
	Frog	0.8348	0.8364	0.7615	0.775	NULL	0.760239
	CuSha	0.933444	0.931491	0.976697	0.892334	NULL	0.977202
	Medusa	0.737698	0.725534	0.736773	0.534144	NULL	0.583266
	GunRock	0.333467	0.332244	0.332134	0.334056	0.332594	0.332982
SSSP	TOTEM		0.528717	0.5644	0.512296	NULL	0.479668
	Frog	0.8348	0.8364	0.7615	0.775	NULL	0.760239
	CuSha	0.933199	0.933106	0.976535	0.893147	NULL	0.980656
	Medusa	0.759679	0.682194	0.766345	0.559669	NULL	0.568269
	GunRock	0.324806	0.322657	0.325463	0.326589	0.325844	0.322851
CC	TOTEM		0.647757	0.622963	0.557834	NULL	0.56812
	Frog	0.8348	0.8364	0.7615	0.775	NULL	0.760239
	CuSha	0.932829	0.933198	0.976602	0.892427	NULL	0.980411
	GunRock	0.371246	0.371149	0.370246	0.371415	0.36888	0.368911

Note: NULL means that the system cannot process such dataset.

Table 8. The Ratio of the Used Global Load Throughput to the System Global Load Throughput

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	49.28	47.7	48.32	38.37	NULL	43.41
	Frog	46.11	45.29	41.95	42.57	NULL	42.52
	CuSha	81.35	81.64	84.55	90.85	NULL	87.75
	Medusa	79.03	78.28	85.63	67.24	NULL	69.74
	GunRock	61.82	62.12	70.32	62.21	56.03	65.03
PR	TOTEM	42.57	48.36	39.64	38.47	NULL	44.96
	Frog	46.11	45.29	41.95	42.57	NULL	42.52
	CuSha	81.36	81.65	84.55	90.85	NULL	87.75
	Medusa	58.76	60.42	69.09	73.26	NULL	71.54
	GunRock	80.31	80.18	80.09	80.09	80.09	80.1
SSSP	TOTEM	19.7	18.43	15.45	17.92	NULL	13.84
	Frog	46.11	45.29	41.95	42.57	NULL	42.52
	CuSha	81.34	81.65	84.55	90.85	NULL	87.75
	Medusa	52.91	53.99	64.7	52.03	NULL	52.16
	GunRock	73.19	69.29	75.59	72.02	74.39	70.88
CC	TOTEM	41.27	45.63	41.5	38.39	NULL	45.89
	Frog	46.11	45.29	41.95	42.57	NULL	42.52
	CuSha	81.35	81.64	84.55	90.85	NULL	87.75
	GunRock	50.71	49.79	64.31	64.78	65.52	64.79

Note: NULL means that the system cannot process such dataset.

Table 9. Average Number of Instructions Executed by Each Warp

Algorithm	System	Amazon	DBLP	RoadNet-CA	Wiki-Talk	Twitter	YouTube
BFS	TOTEM	271	226	296	267	NULL	123
	Frog	706	932	611	724	NULL	417
	CuSha	2,900	2,842	674	353	NULL	333
	Medusa	3,933	5,226.1	3,940.7	4,079.8	NULL	2,251
	GunRock	74	73	74	73	74	74
PR	TOTEM		993	752	734	NULL	767
	Frog	706	932	611	724	NULL	417
	CuSha	2,900	2,842	675	353	NULL	333
	Medusa	2,270	2,986	3,772	1,978	NULL	2,470
	GunRock	409	408	408	408	409	408
SSSP	TOTEM		278	487	261	NULL	159
	Frog	706	932	611	724	NULL	417
	CuSha	2,900	2,842	674	353	NULL	333
	Medusa	956	1,293	1,665	1878	NULL	1,000
	GunRock	100	94	95	98	95	85
CC	TOTEM		285	385	352	NULL	186
	Frog	706	932	611	724	NULL	417
	CuSha	2,900	2,843	674	353	NULL	333
	GunRock	112	108	109	109	108	109

Note: NULL means that the system cannot process such dataset.

5.2.4 Miscellaneous. Table 9 shows the average number of instructions executed by each warp, which can help us understand the activity of the SMs. High variations in the number of executed instructions by every warp indicate the workload of the blocks is in a non-uniform pattern. As analyzed in Section 4.5, high variations of the number of instructions per warp (IPW) occurs when the conditional blocks are executed. A low average IPW indicates there is little variation across the SM, and the compute resources are used inefficiently, while a high average IPW indicates the blocks are in a nonuniform pattern. Table 9 shows the average IPW of GunRock is almost the same with all kinds of algorithms and datasets, which indicates GunRock is better in making use of the computing resource than the other systems. The table also shows that GunRock has the least average number of instructions executed by each warp, which implies there is less branch divergence in GunRock. As we mention in Section 2.2, all the threads in a warp execute the same instructions in a cycle, and the access to the on-device memory of GPU usually has a long latency. So, we need to avoid branch divergence and improve the utilization efficiency of the constant memory in GPU processing systems.

6 CONCLUSIONS AND OPPORTUNITIES

In this article, we surveyed a number of GPU-based graph-processing systems and discussed the challenges inherent in processing graph applications. Some of the challenges also exist in general big data processing and parallel computing. Graph computations are usually data-driven. The graphs have an irregular structure. The size of large-scale graphs may exceed the space memory of a single machine, from which challenges arise as to achieve adequate data locality and parallelization for graph processing.

In order to summarize the performance of major existing graph-processing systems, we apply a taxonomy to classify various GPU-based graph-processing systems. The taxonomy characterizes four aspects of graph processing, including data layout, memory access pattern, workload mapping, and GPU programming. Through the extensive survey of existing systems, we find that most systems did not take drastically different approaches, but added complementary features to the then state-of-the-art techniques. Many systems are similar from the top-level perspective, but differ in their implementation details. There does not seem to be a universally superior combination of features in the existing systems.

Finally, some new research challenges and opportunities for graph processing on GPUs are summarized as follows:

— **Graph processing on hybrid systems**. As the GPU memory is limited, how to use GPU processing large-scale graphs is another major challenge. Nowadays, *GPU-enabled clouds, GPU clusters, and CPU/GPU hybrid systems*, which have large memory space, are widely used in various applications. Porting graph-processing algorithms to these systems is a promising research direction for large-scale graph processing. Graph partitioning and workload mapping are the fundamental challenges for graph processing on such systems. First of all, we need to partition the graph and the corresponding processing workload onto the GPUs and CPUs. Uneven workload mapping can lead to load imbalance in the system. Since vertices in a big graph can be connected with a complex pattern, how to partition the graph to achieve load balancing is a challenging problem. Second, a single GPU memory is limited; utilizing GPU memory efficiently plays an important role in achieving good graph-processing performance, for which a good memory access pattern is another essential aspect. Existing works are in general going toward this direction. However, how to capitalize the advantage of the GPU architecture for efficient graph processing remains a challenge, which requires the programmers to make bespoke efforts for the graph applications in question.

Moreover, the limitation of GPU memory impedes the processing of large-scale graphs. Consequently, for the CPU/GPU hybrid systems, out-of-core data management techniques are required to tackle the memory overflow problem in GPU when the size of a graph exceeds the capacity of the GPU memory. Unfortunately, this issue has not been addressed sufficiently, to the best of our knowledge. In addition, the strategy of data partitioning between CPU memory and GPU memory can also substantially affect the quality and efficiency of graph processing when out-of-core techniques are applied. GPU graph-processing systems need to consider whether solely using the GPU memory or moving some graph data (if so how much) to the CPU memory in a multi-node environment. New GPU architectures such as 3D stacked memory in newer GPU devices can provide another alternative solution.

— **Graph processing on new GPU architecture**. Developing a graph-processing system is a systematic project in the sense that it needs to strike a balance among many important factors in graph processing. Besides the three main aspects summarized above, there are other challenges as well, such as benchmark setting, branch divergence, communication, and so on. Some of these challenges, such as branch divergence, are caused by the complex programming model on GPU. A fundamental solution to these challenges is to develop a more flexible and easy-to-use programming API for GPU. GunRock provides a good example in this direction. As for the challenges related to the communications in GPU processing, new features such as Unified Memory, NVLink, and 3D stacked memory may offer the solutions to this issue. By using the unified memory, programmers can be liberated from the task of complex memory allocation. With the support of NVLink, a GPU device can communicate with a CPU and other GPUs directly via high-bandwidth connections. Using 3D stacked memory can expand the GPU memory by multiple folds. Consequently, larger-scale graphs can then be loaded into the GPU memory all at once, eliminating the need of out-of-core executions on GPU.

— **Dynamic graph processing on GPUs**. Dynamic graph is an important application in the real world, but there is little work on GPU-based dynamic graph processing. So, designing and implementing systems to support dynamic graph processing on GPUs is another interesting and challenging research direction. In dynamic graph processing, the graph structure can be updated frequently in runtime. This poses additional challenges to designing data layout and achieving good memory access patterns. Furthermore, how to dynamically maintain balanced workload mapping with the rapid changes of graphs is highly challenging.

— **Machine-learning applications**. Graph-processing systems are also widely adopted in training large machine-learning models. A highly interesting and potentially influential research direction is to identify the properties of machine-learning applications and build specialized GPU-based graph-processing algorithms or systems to enhance the performance of machine-learning applications.

ACKNOWLEDGMENTS

We thank Beng Chin Ooi for his valuable comments and the anonymous reviewers for the valuable feedback on earlier versions of this article.

REFERENCES

2014. BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs. *J. Parallel Distrib. Comput.* 74, 7 (2014), 2639–2647. DOI: https://doi.org/10.1016/j.jpdc.2014.03.002

Gharaibeh Abdullah, Reza Tahsin, Santos-Neto Elizeu, Costa Lauro, Beltro, Sallinen Scott, and Ripeanu Matei. 2014. *Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems.* Technical Report. Networked Systems Lab, University of British Columbia. 1–36.

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

AMD AMD. 2011. Accelerated parallel processing: OpenCL programming guide. Retrieved from http://developer.amd.com/appsdk.

Prabhu Arjun, Kilgariff Emmett, and M. Wittenbrink Craig. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 2 (2011), 50–59. DOI: https://doi.org/doi.ieeecomputersociety.org/10.1109/MM.2011.24

Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67. DOI: https://doi.org/10.1145/1562764.1562783

Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 781–792. DOI: https://doi.org/10.1109/SC.2014.69

Ariful Azad, Aydin Bulu, and Alex Pothen. 2015. A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS'15)*. 1075–1084. DOI: https://doi.org/10.1109/IPDPS.2015.68

Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, 105–114. DOI: https://doi.org/10.1145/1693453.1693470

Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, Article 18, 11 pages. DOI: https://doi.org/10.1145/1654059.1654078

Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In *Proceedings of the 2016 45th International Conference on Parallel Processing (ICPP'16)*. 496–505. DOI: https://doi.org/10.1109/ICPP.2016.64

Christian Bienia and Kai Li. 2010. Fidelity and scaling of the PARSEC benchmark inputs. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization (IISWC'10)*. 1–10. DOI: https://doi.org/10.1109/IISWC.2010.5649519

Guy E. Blelloch. 1990. *Vector Models for Data-parallel Computing*. Vol. 356. MIT Press, Cambridge.

John Adrian Bondy and Uppaluri Siva Ramachandra Murty. 1976. *Graph Theory with Applications*. Elsevier Science Ltd/North-Holland, New York.

Nicolas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 49–60.

Aydn Bulu, John R. Gilbert, and Ceren Budak. 2010. Solving path problems on the GPU. *Parallel Comput.* 36, 56 (2010), 241–253. DOI: https://doi.org/10.1016/j.parco.2009.12.002

J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. 2015. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE'15)*. 1131–1142. DOI: https://doi.org/10.1109/ICDE.2015.7113362

Aleksandar Colic, Hari Kalva, and Borko Furht. 2010. Exploring NVIDIA-CUDA for video coding. In *Proceedings of the 1st Annual ACM SIGMM Conference on Multimedia Systems (MMSys'10)*. ACM, New York, 13–22. DOI: https://doi.org/10.1145/1730836.1730839

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*. ACM, New York, 1–12. DOI: https://doi.org/10.1145/155332.155333

A. Davidson, S. Baxter, M. Garland, and J. D. Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*. 349–359. DOI: https://doi.org/10.1109/IPDPS.2014.45

Erich Elsen and Vishal Vaidyanathan. 2013. A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction. Retrieved from https://github.com/RoyalCaliber/vertexAPI2.

Bas O. Fagginger Auer and Rob H. Bisseling. 2012. *A GPU Algorithm for Greedy Graph Matching*. Springer, Berlin, 108–119. DOI: https://doi.org/10.1007/978-3-642-30397-5_10

W. Fang, B. He, Q. Luo, and N. K. Govindaraju. 2011. Mars: Accelerating MapReduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.* 22, 4 (April 2011), 608–620. DOI: https://doi.org/10.1109/TPDS.2010.158

Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Software* 43, 4, Article 30 (Jan. 2017), 49 pages. DOI: https://doi.org/10.1145/3017994

Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. 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 (GRADES'14)*. ACM, New York, Article 2, 6 pages. DOI: https://doi.org/10.1145/2621934.2621936

W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. 407–420. DOI: https://doi.org/10.1109/MICRO.2007.30

Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, 345–354. DOI: https://doi.org/10.1145/2370816.2370866

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 17–30.

Sudipto Guha, Andrew McGregor, and David Tench. 2015. Vertex and hyperedge connectivity in dynamic graph streams. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'15)*. ACM, New York, 241–247. DOI: https://doi.org/10.1145/2745754.2745763

Udo Hahn and Ulrich Reimer. 1984. Computing text constituency: An algorithmic approach to the generation of text graphs. In *Proceedings of the 7th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'84)*. British Computer Society, Swinton, UK, 343–368.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. DOI: https://doi.org/10.1145/1656274.1656278

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, New York, 77–85. DOI: https://doi.org/10.1145/2487575.2487581

Pawan Harish and P. J. Narayanan. 2007. *Accelerating Large Graph Algorithms on the GPU Using CUDA*. Springer, Berlin, 197–208. DOI: https://doi.org/10.1007/978-3-540-77220-0_21

Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, 260–269. DOI: https://doi.org/10.1145/1454115.1454152

Guoming He, Haijun Feng, Cuiping Li, and Hong Chen. 2010. Parallel SimRank computation on large graphs with iterative aggregation. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'10)*. ACM, New York, 543–552. DOI: https://doi.org/10.1145/1835804.1835874

Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endowment* 8, 4 (Dec. 2014), 329–340. DOI: https://doi.org/10.14778/2735496.2735497

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, 349–362. DOI: https://doi.org/10.1145/2150976.2151013

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011a. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, 267–276. DOI: https://doi.org/10.1145/1941553.1941590

Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011b. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 78–88. DOI: https://doi.org/10.1109/PACT.2011.14

Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart. 2011. Edge v. node parallelism for graph centrality metrics. *GPU Comput. Gems Jade Ed.* 2 (2011), 15–28.

M. I. Jordan and T. M. Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.

Gary J. Katz and Joseph T. Kider, Jr. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH'08)*. Eurographics Association, Aire-la-Ville, Switzerland, 47–55.

Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, 169–182. DOI: https://doi.org/10.1145/2465351.2465369

Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, 239–252. DOI: https://doi.org/10.1145/2600212.2600227

Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York, 447–461. DOI: https://doi.org/10.1145/2882903.2915204

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 31–46. Retrieved from https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola.

Edward Ashford Lee and David G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* 100, 1 (1987), 24–35.

Jure Leskovec. 2009. Stanford Network Analysis Project. Retrieved November 29, 2016 from http://snap.stanford.edu/index.html.

Da Li and Michela Becchi. 2013. Deploying graph algorithms on GPUs: An adaptive solution. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS'13)*. 1013–1024. DOI: https://doi.org/10.1109/IPDPS.2013.101

Kenli Li, Wangdong Yang, and Keqin Li. 2015. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Trans. Parallel Distrib. Syst.* 26, 1 (Jan. 2015), 196–205. DOI: https://doi.org/10.1109/TPDS.2014.2308221

Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, Article 68, 12 pages. DOI: https://doi.org/10.1145/2807591.2807594

Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent breadth-first search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York, 403–416. DOI: https://doi.org/10.1145/2882903.2882959

Peter J. Lu, Hidekazu Oki, Catherine A. Frey, Gregory E. Chamitoff, Leroy Chiao, Edward M. Fincke, C. Michael Foale, Sandra H. Magnus, William S. McArthur, Daniel M. Tani, Peggy A. Whitson, Jeffrey N. Williams, William V. Meyer, Ronald J. Sicker, Brion J. Au, Mark Christiansen, Andrew B. Schofield, and David A. Weitz. 2010. Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the international space station. *J. Real-Time Image Process.* 5, 3 (2010), 179–193. DOI: https://doi.org/10.1007/s11554-009-0133-1

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, 135–146. DOI: https://doi.org/10.1145/1807167.1807184

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL'14)*. Association for Computational Linguistics, 55–60.

Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, 235–246. DOI: https://doi.org/10.1145/1815961.1815992

Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, 117–128. DOI: https://doi.org/10.1145/2145816.2145832

George A. Miller. 1995. WordNet: A lexical database for english. *Commun. ACM* 38, 11 (Nov. 1995), 39–41. DOI: https://doi.org/10.1145/219717.219748

Ioannis Mitliagkas, Michael Borokhovich, Alexandros G. Dimakis, and Constantine Caramanis. 2015. FrogWild!: Fast PageRank approximations on graph engines. *Proceedings of the VLDB Endowment* 8, 8 (2015), 874–885.

Alexander Monakov and Arutyun Avetisyan. 2009. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*, Koen Bertels, Nikitas Dimopoulos, Cristina Silvano, and Stephan Wong (Eds.). Springer, Berlin, 289–297. DOI: https://doi.org/10.1007/978-3-642-03138-0_32

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, 308–317. DOI: https://doi.org/10.1145/2155620.2155656

P. J. Narayanan, Kothapalli Kishore, and Jyothish Soman. 2010. A fast GPU algorithm for graph connectivity. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. 1–8. DOI: https://doi.org/10.1109/IPDPSW.2010.5470817

Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, 147–156. DOI: https://doi.org/10.1145/2442516.2442531

NVIDIA. 2009. NVIDIA Fermi Architecture Whitepaper. Retrieved from http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

NVIDIA. 2012. Kepler Compute Architecture White Paper. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

NVIDIA. 2014. NVIDIA GeForce GTX 750 Ti-Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt. Retrieved from http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

NVIDIA. 2016a. CUDA C Programming Guide. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide#introduction.

NVIDIA. 2016b. NVIDIA Tesla P100—The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the Worlds Fastest GPU. Retrieved from http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* 26, 1 (2007), 80–113. DOI: https://doi.org/10.1111/j.1467-8659.2007.01012.x

Greta L. Polites and Richard T. Watson. 2008. The centrality and prestige of CACM. *Commun. ACM* 51, 1 (Jan. 2008), 95–100. DOI: https://doi.org/10.1145/1327452.1327454

Matthew Richardson and Pedro M. Domingos. 2001. The intelligent surfer: Probabilistic combination of link and content information in PageRank. In *Proceedings of the the Conference and Workshop on Neural Information Processing Systems (NIPS'01)*. 1441–1448.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, 472–488. DOI: https://doi.org/10.1145/2517349.2522740

Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast PageRank computation on a GPU cluster. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'12)*. 450–456. DOI: https://doi.org/10.1109/PDP.2012.78

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27, 3, Article 18 (Aug. 2008), 15 pages. DOI: https://doi.org/10.1145/1360612.1360617

Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, Article 28, 12 pages. DOI: https://doi.org/10.1145/2807591.2807655

Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. 2015. GStream: A graph streaming processing method for large-scale graphs on GPUs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, 253–254. DOI: https://doi.org/10.1145/2688500.2688526

Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. 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'15)*. ACM, New York, 271–272. DOI: https://doi.org/10.1145/2688500.2688542

Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, 353–364. DOI: https://doi.org/10.1145/2304576.2304624

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. DOI: https://doi.org/10.1145/79173.79181

Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. 2009. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG'09)*. ACM, New York, 167–171. DOI: https://doi.org/10.1145/1572769.1572796

Vibhav Vineet and P. J. Narayanan. 2008. CUDA cuts: Fast graph cuts on the GPU. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'08)*. 1–8. DOI: https://doi.org/10.1109/CVPRW.2008.4563095

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, New York, Article 11, 12 pages. DOI: https://doi.org/10.1145/2851141.2851145

Douglas Brent West. 2001. *Introduction to Graph Theory*. Prentice Hall, New Jersey.

Bin Wu and Yahong Du. 2010. Cloud-based connected component algorithm. In *Proceedings of the 2010 International Conference on Artificial Intelligence and Computational Intelligence (AICI'10)*, Vol. 3. 122–126. DOI: https://doi.org/10.1109/AICI.2010.360

Nan Wu, Bin Li, Hua Wang, ChengWen Xing, and JingMing Kuang. 2015. Distributed cooperative localization based on Gaussian message passing on factor graph in wireless networks. *SCIENCE CHINA Inf. Sci.* 58, 4 (2015), 1–15. DOI: https://doi.org/10.1007/s11432-014-5172-y

Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient pagerank and SPMV computation on amd GPUS. In *Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP'10)*. IEEE, 81–89.

Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, 107–118. DOI: https://doi.org/10.1145/2555243.2555255

Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO'17)*. IEEE, 27–38. http://dl.acm.org/citation.cfm?id=3049832.3049836

Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. 2016. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16)*. 912–920. DOI: https://doi.org/10.1109/IPDPSW.2016.185

Jianlong Zhong and Bingsheng He. 2013. Towards GPU-accelerated large-scale graph processing in the cloud. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01 (CLOUDCOM'13)*. IEEE Computer Society, 9–16. DOI: https://doi.org/10.1109/CloudCom.2013.8

Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1543–1552. DOI: https://doi.org/10.1109/TPDS.2013.111

Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, 375–386.

Received February 2017; revised June 2017; accepted July 2017