Frog: Asynchronous Graph Processing on GPU with Hybrid Coloring Model

Xuanhua Shi, *Member, IEEE*, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, *Member, IEEE*, Bingsheng He, *Member, IEEE*, and Hai Jin, *Senior Member, IEEE*

Abstract—GPUs have been increasingly used to accelerate graph processing for complicated computational problems regarding graph theory. Many parallel graph algorithms adopt the asynchronous computing model to accelerate the iterative convergence. Unfortunately, the consistent asynchronous computing requires locking or atomic operations, leading to significant penalties/overheads when implemented on GPUs. As such, the coloring algorithm is adopted to separate the vertices with potential updating conflicts, guaranteeing the consistency/correctness of the parallel processing. Common coloring algorithms, however, may suffer from low parallelism because of a large number of colors generally required for processing a large-scale graph with billions of vertices. We propose a light-weight asynchronous processing framework called Frog with a preprocessing/hybrid coloring model. The fundamental idea is based on the Pareto principle (or 80-20 rule) about coloring algorithms as we observed through masses of real-world graph coloring cases. We find that a majority of vertices (about 80 percent) are colored with only a few colors, such that they can be read and updated in a very high degree of parallelism without violating the sequential consistency. Accordingly, our solution separates the processing of the vertices based on the distribution of colors. In this work, we mainly answer three questions: (1) how to partition the vertices in a sparse graph with maximized parallelism, (2) how to process large-scale graphs that cannot fit into GPU memory, and (3) how to reduce the overhead of data transfers on PCIe while processing each partition. We conduct experiments on real-world data (Amazon, DBLP, YouTube, RoadNet-CA, WikiTalk, and Twitter) to evaluate our approach and make comparisons with well-known non-preprocessed (such as Totem, Medusa, MapGraph, and Gunrock) and preprocessed (Cusha) approaches, by testing four classical algorithms (BFS, PageRank, SSSP, and CC). On all the tested applications and datasets, Frog is able to significantly outperform existing GPU-based graph processing systems except Gunrock and MapGraph. MapGraph gets better performance than Frog when running BFS on RoadNet-CA. The comparison between Gunrock and Frog is inconclusive. Frog can outperform Gunrock more than 1.04X when running PageRank and SSSP, while the advantage of Frog is not obvious when running BFS and CC on some datasets especially for RoadNet-CA.

Index Terms—GPGPU, graph processing, asynchronous computing model

1 Introduction

Graph is a fundamental data representation used in many large-size and complicated problems. Large graph processing can be commonly found in many computational domains, such as social networks and web link analysis. In such domains, it is critical to develop a light-weight approach that can process large graphs with millions/billions of vertices/edges very efficiently.

GPUs are often used to accelerate the CUDA based large scale computing, not only because of the massive computation power, but also due to much higher memory bandwidth and throughput. In recent years, some researchers try to use GPU to accelerate graph computing, but there are still some issues in the existing solutions for GPU-based graph processing.

- The maximum size of available GPU memory is still very limited for large graph processing. For example, there is only 12 GB of memory for NVIDIA Tesla K40. As such, a lot of existing works [1], [2], [3] make use of a hybrid model with both CPU and GPU to execute large-size graphs. There are two general ideas while partitioning and processing large-scale graphs. One is partitioning the large graph into small pieces and offloading them one by one to the GPU for the execution. The other one is placing some vertices in one type of processor (such as GPU cores) and putting the rest in the other type (such as CPU), so as to make use of both types of compute resources. The CPU, however, is still a bottleneck when processing large-scale graphs. Hence, it is still necessary to design a framework that can process the entire graph on GPU processors.
- Most existing GPU-accelerated graph frameworks (such as Totem [2]) are designed based on the synchronous processing model—Bulk Synchronous

- *X. Shi, X. Luo, J. Liang, P. Zhao, and H. Jin are with the Services Computing Technology and System Lab, Big Data Technology and System Lab, School of Computer, Huazhong University of Science and Technology, Wuhan 430074, China.*
  E-mail: {xhshi, luoxuan, junlingliang, zhaopeng, hjin}@hust.edu.cn.
- *S. Di is with the Argonne National Laboratory, Lemont, IL 60439.*
  E-mail: sdi1@anl.gov.
- *B. He is with the Department of Computer Science, National University of Singapore, Singapore 119077. E-mail: hebs@comp.nus.edu.sg.*

Manuscript received 11 Dec. 2015; revised 4 Aug. 2017; accepted 21 Aug. 2017. Date of publication 29 Aug. 2017; date of current version 5 Dec. 2017.
(Corresponding author: Xuan Luo.)
Recommended for acceptance by E. Pitoura.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2745562

Parallel (BSP) model [4]. The BSP model divides the data processing procedure into several *super-steps*, each of which consists of three phases: *computation*, *communication* and *synchronization*. Such a model, however, will introduce a huge cost in synchronization especially as the graph size grows significantly, because any message processing must be finished in the previous super-step before moving to the next one.

- In comparison to synchronous models, there are some asynchronous models that have been proved more efficient in processing graphs [5], but they are not very suitable for parallel graph processing on GPU. In asynchronous models, more vertices can be updated simultaneously in each step, thus with a very huge performance gain on the graph processing. In order to ensure the correct/consistent processing results in the parallel computations, many existing solutions (such as GraphLab [5]) adopt fine-grained locking protocols or update most vertices sequentially for simplicity. Locking policy, however, is unsuitable for GPU-based parallel processing because of the huge cost of the locking operations on GPU.

In our work, we design a lock-free parallel graph processing method named Frog with a graph coloring model. In this model, each pair of adjacent vertices with potential update conflicts will be colored differently. The vertices with the same colors are allowed to be processed in parallel, also guaranteeing the sequential consistency [5] of the parallel execution.

Graph coloring issue [6] is an NP-complete problem, which means that it is impossible to find the optimal solution with polynomial time unless P=NP. As for a large graph with billions of vertices, existing graph coloring heuristics are, in general, not viable to use because coloring graphs with billions of vertices may raise hundreds of different colors, which will lead to significant overhead when processing such hundreds of super-steps by a round-robin scheduler.

Our design is motivated by a crucial observation that the whole course of graph coloring usually follows a Pareto principle (or 80-20 rule). That is, a large majority of vertices (roughly 80 percent) are colored with only a small number of colors (about 20 percent or less), while only a small amount of vertices (about 20 percent) are assigned with a large number of different colors (about 80 percent). Based on such a finding, our solution will process the vertices based on their coloring distributions. In particular, we process a majority of the mutually non-adjacent vertices with the same colors in a high degree of parallelism on GPU, and process the minority of the "conflict" vertices in a separate super-step.

There are still three issues in designing such a parallel model based on coloring distribution, which will be our focus. In this paper, we mainly answer the following three questions:

- What is the difference between synchronous and asynchronous graph processing models on GPU(s)?
- Given a small number of colors, how to partition a large graph and select 80 percent of vertices that are assigned different colors?
- How to reduce the overhead of data transfers on PCIe while processing each partition?

Overall, we have the following four contributions to address the above problems.

1) *An efficient hybrid graph coloring algorithm*: We propose a relaxed pre-partition method that can solve the problem of vertex classification using a moderate number of colors. This method does not force all adjacent vertices to be assigned with different colors, which is particularly different from other graph coloring algorithms. Instead, we only ensure that there are no adjacent vertices assigned together into the small set of colors. For the vertices with the rest of colors, we combine them together into one color and process them in a super-step.
2) *A coloring-based asynchronous execution model on GPUs*: We design and implement an execution engine, in order to scan and update the graph that is partitioned by our hybrid graph coloring algorithm. The partitioned graph will be scanned color by color, and all vertices with the same color will be updated in parallel (one color as per kernel[1] execution). Since concurrently accessed vertices must be non-adjacent, there will be no need to lock the adjacent edges when updating a vertex.
3) *A light-weight streaming execution engine for handling the large graph on GPUs*: In our design, when processing each partition, the data transfers are overlapped with the executions of GPU kernel functions, so as to minimize the overhead of PCIe data transfers.
4) *An open source toolkit for graph processing on GPUs*: We carefully implemented the whole Frog toolkit, which is available to download for free.[2] We evaluate our hybrid graph coloring model by comparing to five other state-of-the-art related systems. Experiments show that our solution obtains great performance gains over three systems (Cusha [7], Medusa [8] and Totem [2]). Except for RoadNet-CA on BFS, Frog gets better performance than MapGraph [9]. Frog outperforms Gunrock [10] by 4.51X-20.18X on PageRank , and 1.04X-2.13X on SSSP. For BFS and CC, Gunrock shows better performance especially for RoadNet-CA. Frog outperforms Gunrock by 0.34X-1.29X on BFS, and 0.13X-3.32X on CC.

The rest of this paper is organized as follows. In Section 2, we present some background and the motivation of this work. We will then present our novel and relaxed graph coloring algorithm in Section 3. Based on this partitioning strategy, we will present the overview of our system and our asynchronous approach on handling large-size graphs in Section 4. The results of performance evaluation will be presented and analyzed in Section 5. Related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2 Background and Motivation

In this section, we first introduce some characteristics of GPU processors. Subsequently, we explain the sequential

[1] *Kernel* is referred to the procedure executed on GPU.

[2] Source codes and technical report (including program APIs, intensive performance evaluations) are available at http://grid.hust.edu.cn/xhshi/projects/frog.html

consistency in parallel programming. Finally, we present the motivation of this work. We discuss why we choose asynchronous execution model over synchronous model to process graphs on GPU, why we adopt graph coloring as the basic method, and the pros and cons of existing graph coloring methods.

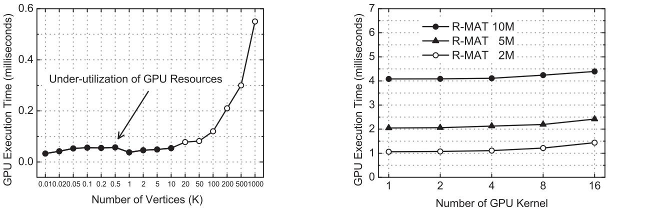
2.1 Characteristics of GPU Processors

Today's GPUs have a high computation power with massive multithreading and become popular in graph processing due to the NVIDIA CUDA framework, and have been used widely to accelerate graph processing. Although GPU is a fairly powerful computing processor, it suffers from some limitations. First, GPU has very limited global memory. The PCIe bus, as the bridge of data transfer between CPU and GPU memory, may become a bottleneck because of the limited bandwidth. Usually, PCIe transfers consume a large fraction (30 to 80 percent) of total execution time dedicated to CPU-GPU memory transfer and GPU kernel execution. This requires an efficient method to enforce data transfers between these two different devices. Second, some general operations used on CPU-based systems are very costly for GPU-based processing. Lock operations and atomics, for example, are usually very expensive on massively parallel GPUs and often result in the critical path in a program. This is one reason why we design an efficient coloring based execution instead of using locks on GPUs. Third, it is difficult to write correct and efficient GPU programs and even more difficult for graph processing. There are some existing studies to make easy programming on GPUs [7], [8], [9], [10].

Our system Frog is aimed at solving the above problems. Considering the limited global memory, a large-scale graph will be divided by our coloring algorithm, such that it can be processed separately in different partitions. In order to reduce the overhead of locking mechanism, our coloring algorithm ensures the asynchronous execution of each partition. Our system also provides APIs for developers to make the programming easier. So the developers can deal with graph processing without concerning the details of GPU programming model.

Not only do we consider GPU existing limitations, but we also find two significant characteristics of GPU processors based on our experiments. We conduct these experiments using synthetic graphs generated by GTgraph [11], which can generate graphs with power-law degree distributions (or Pareto principle) and small-world characteristics based on Recursive MATrix (RMAT) model [12]. Each R-MAT graph in our experiments has n vertices and $10 * n$ edges. We are mostly concerned about the performance of processing power-law graphs, since such graphs have been widely studied in many domains.

One characteristic is that the execution performance of GPU is closely related to the number of vertices to process. In order to keep high execution efficiency, one should make sure that a large number of vertices will be updated in parallel by thousands of threads or more during the graph processing. If the parallelism is not high enough, it would take much longer time to complete the tasks as the GPU computing resources cannot be fully utilized. This is due to the fact that one has to actually assign work to all the pipelines on GPU. Fig. 1a shows that it costs almost the same time when processing graphs with 10 vertices, 100 vertices, 1k vertices and even 10k vertices on GPU. The kernel code was to calculate the out-degree of vertices and each vertex was updated by one thread. Obviously, we should have 14K or more vertices/threads run concurrently on NVIDIA K20m GPUs, in order to obtain a good performance.



(a) GPU execution time with different scale R-MAT graphs

(b) GPU execution time with different number of GPU kernels

Fig. 1. GPU execution time with different scale R-MAT graphs and different number of GPU kernels. (a) It takes almost the same time when processing graphs with 10 vertices and 10k vertices, which means this scale of graphs can not utilize the GPU resources. (b) It takes almost the same time while processing the same graph by different number of kernels, with only a little overhead of scheduling between different kernels.

The other characteristic is that the overhead of scheduling between different kernels is low. NVIDIA CUDA is a scalable parallel programming model and a software environment for parallel computing, so we adopt it as our development platform. CUDA uses thousands of threads that can execute the same kernel to achieve performance gains, and the kernel can be launched very quickly on GPU. Fig. 1b shows that it costs almost the same time while processing the same graph using different numbers of kernels, which suffers only a little overhead of scheduling. That is, the overhead induced with kernel scheme is low even we partition the graph and make kernel calls for each partition.

2.2 Significance and Classification of Sequential Consistency

It is mandatory to take into account the sequential consistency when coding a parallel algorithm on GPU. According to the definition proposed by Lamport, sequential consistency guarantees "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [13]. Sequential consistency ensures that the instructions in each thread will be executed as the programming order. The results would be wrong without sequential consistency, because of the wrong order of the execution and the data race between the parallel threads [14], [15], [16].

There are three data consistency models [15] that can be used to guarantee the sequential consistency in graph processing. Each one of them can guarantee correct results when a parallel algorithm performs update operations in any sequence, with different permissions of reading and writing given by different data consistency models. When the update function is updating a vertex V_p, the weakest vertex consistency model ensures that each update function has only read-write access to the vertex V_p itself and read-only access to its adjacent edges, yet its adjacent vertices cannot be read. The edge consistency model ensures each update function has an exclusive read-write access to the vertex V_p and adjacent edges but read-only access to its adjacent vertices. The strongest full consistency mode ensures read-write access to the vertex V_p and all its adjacent edges and vertices.

An appropriate selection of the consistency model directly affects the correctness of the program. We do not have to support all of the three models in the asynchronous framework. In our system, we adopt the edge consistency model, because it can reach a higher parallelism than the full consistency model and suffer less cost in guaranteeing the data consistency than the vertex consistency model. By comparison, in the vertex consistency model, we need to ensure all updating vertices cannot change immediately values of all edges and vertices, which leads to a lower convergence speed.

2.3 Motivation of This Work

For the purpose of easy programming and system deployment, the synchronous execution model is particularly suitable for the GPU architecture. This is because many graph algorithms need to iteratively update all the vertices and edges which are highly correlated to each other. The algorithm PageRank, for example, updates the rank values of vertices iteratively until majority of rank values do not change clearly any more. In general, when updating some vertex (such as changing the rank values), the algorithm also needs to query the information of its neighborhood (such as the rank values of its neighboring vertices). In other words, some vertex is required to "see" the updates of its neighboring vertices before being updated. To this end, it is fairly straight-forward to adopt synchronous model to solve the graph problems. However, this also results in some significant issues, as listed below.

- *Slow Convergence*: Under the synchronous execution model, a vertex cannot see the updates of its neighboring vertices until the end of an iteration, so the synchronous execution model can only update the vertices based on the values at the previous time step, inevitably causing a slow convergence. By contrast, asynchronous execution model is able to update vertices using the most recent values, which will converge much faster than synchronous model. This has been confirmed by a lot of existing researches [5], [17], [18].
- *Non-negligible Synchronization Overhead*: Synchronous model is fairly suitable to process the graph with a large number of active vertices to update in an iteration step, in that the synchronization overhead like the cost of the global barrier could be negligible compared to the huge updating cost in each iteration. The number of active vertices, however, is likely to change a lot during the execution. For instance, the amount of active vertices varied significantly in different BFS levels [19]. As a consequence, the synchronization overhead could be significant especially when there are only a small number of active vertices in an iteration.
- *Limits on Coordination*: Many algorithms may not converge in the synchronous model [20], [21], such that they cannot resolve the situation with a lot of coordinations among adjacent vertices. More specifically, all adjacent vertices in the synchronous model can only be updated simultaneously according to the same values at previous steps with no coordination. On the other side, these algorithms can converge under the asynchronous model if we ensure the appropriate consistency at the same time [22].
- *Low Adaptability to Load Imbalance*: In order to complete the whole graph-processing work, the overall performance is usually determined by the slowest GPU thread. Since the synchronous model always synchronizes each iteration during the execution, it is hard to perform the load balance in such a short period. By contrast, the asynchronous model is more flexible because of the loosely coupled parallel threads.

Due to above problems of synchronous model, asynchronous model has been paid more and more attentions, but it is rather difficult to design and manage. Specifically, one has to always guarantee the serializability during the whole execution. That is, any correct result of a parallel execution must be equal to the result of some sequential execution. There are several approaches to enforce serializability, such as lock-based protocol [5] and sequential execution [20]. Locking or atomic operations, however, would be very expensive on massively parallel GPUs [23], [24], thus neither locking protocol nor sequentially updating is a viable solution.

In comparison to locking protocol and atomic operations, graph coloring is a more efficient asynchronous model because it can process vertices in parallel and guarantee the serializability meanwhile. Its fundamental principle is to construct a coloring mechanism that assigns each vertex a color such that no adjacent vertices share the same color. This completely conforms to the *edge consistency model* defined by GraphLab [15]. Under the edge consistency model, each update operation on some vertex is allowed to modify the vertex's values as well as its adjacent edges, but it can only read the values of adjacent vertices. Vertices of the same color can be updated in parallel without any locking or atomic protocol. Apparently, the edge consistency model significantly increases parallelism compared to the locking protocol, since the vertices sharing the same edges will be partitioned into different color groups and processed in separate execution phases.

Traditional strict graph coloring, however, may not work very efficiently, especially when the number of vertices/edges is extremely large. For a data graph with billions of vertices, hundreds of colors have to be generated to complete the graph coloring (as shown in Fig. 2). This may result in a large number of small color groups each with only a few vertices to update, which eventually degrades the parallelism significantly.

Based on a lot of experiments (shown as the Fig. 7 in the evaluation section), we observe a typical Pareto principle in the coloring process, i.e., a large number of vertices are colored with only a few colors. This inspires us to design a graph coloring algorithm based on the coloring distribution, significantly improving the parallelism for the graph processing on GPU.

3 Coloring Based Graph Partitioning

In this section, we describe the features required by coloring based graph partitioning strategies, as well as the detailed

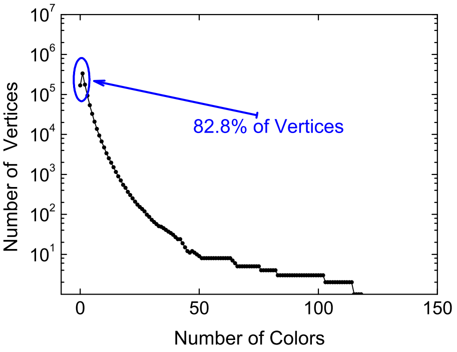


Fig. 2. Result of coloring on graph DBLP. Strict graph coloring algorithm requires 119 colors. The top four colors cover 82.8 percent of vertices, while the remaining hundreds of colors cover only 17.5 percent of vertices.

coloring method. Our partitioning strategy processes graphs iteratively in three stages: 1) loading edges in the form of (v_{src}, v_{dst}) from the file on disk; 2) putting the vertex v_{src} in one of partitions; and 3) putting the vertex v_{dst} in another different partition. Such three stages are explained in detail below, using a concrete example.

3.1 Required Features for Coloring Based Graph Partitioning Strategies

We propose the following characteristics that are supposed to be taken into account by the coloring based graph partitioning strategies.

- *Low time complexity:* It will be more and more costly to process a graph as its size grows. Since the partitioning is just the preliminary step, our partitioning method should not lead to a long analysis time.
- *High Reusability:* When running some applications with the same graph, we should avoid performing the partitioning step redundantly, because the partition result is supposed to be unchanged for all cases with the same graph.
- *Minimized Data Transfer Overhead:* Since it is always costly to scan a disk file that stores the graph data, we should make sure that the file loading only occurs when necessary (i.e., to make sure each processing just scans the original data file only once).

3.2 Coloring-Based Partitioning Strategy

We color the vertices of the given graph using a small number of colors, to make sure that each partition contains a sufficient number of vertices that can be processed in a high degree of parallelism. Suppose we will generate n partitions, denoted as p_1, p_2, p_3, ..., p_n. For the first $n-1$ partitions (p_1, p_2, ..., p_{n-1}), all the vertices in the same partition share the same color and satisfy the edge consistency model, which means that no two adjacent vertices share the same color. The last partition p_n is a hybrid partition whose vertices are allowed to be assigned different colors.

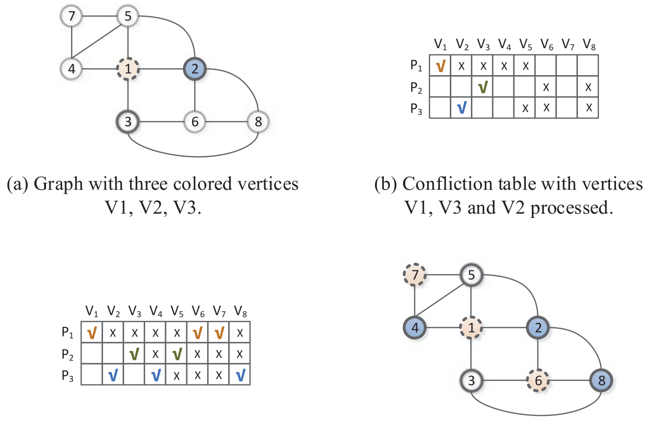
In comparison to classic vertex coloring approaches for maintaining consistency, our algorithm has three distinct features.

1) *Maximizing the Degree of Parallelism for first $n-1$ partitions*: Instead of guaranteeing that all of the adjacent vertices are assigned different colors in the whole graph, we adopt a relaxed hybrid k-coloring method, which makes sure that a majority of adjacent vertices are assigned different colors. That is, our solution allows a few pairs of adjacent vertices (about 20 percent of vertices) to be assigned the same color, such that the whole graph can be processed in a high degree of parallelism for processing most of vertices. In contrast, if we adopt a full coloring strategy, there may be some particular colors that only involve a small number of vertices. This will be validated in Section 5.2 later on. When the system processes the minority of vertices corresponding to such colors, we may not fully harness the GPUs parallelism power and processing ability. Our algorithm can avoid such a low resource utilization issue since we maximize the number of vertices in each of the first $n-1$ color partitions/chunks.
2) *Guaranteeing the Consistency for nth partition*: Our solution has a mix color partition (i.e., the last partition) wherein the adjacent vertices could be assigned different colors. This is why we call our coloring algorithm a *hybrid* method. We adopt GPU atomic operations to process the vertices in this partition, in order to guarantee the consistency.
3) *Low Complexity Due to High Reusability*: Our solution has a very low complexity, since we try to reuse the previous analysis of the graph in the following coloring process. In particular, our algorithm only takes $O(v+e)$ time to finish the process. If the target graph is added by some new vertices and edges without changing the topological structure of origin data graph, our solution does not have to perform all of the coloring operations repeatedly but just conducts some necessary extra operations compared to the previous analysis, so as to minimize the time complexity in graph processing.

In the following text, we first present our coloring method, and then describe it using a concrete example.

Our partitioning strategy processes graphs iteratively in three stages.

- *Stage 1:* We create a table to record the coloring results. The size of this table is $n \cdot |V|$, where n is the number of partitions and $|V|$ represents the number of vertices of the graph (assume we know this number before processing the graph). Then, we read edges of the graph one by one throughout the data file and denote them in the form of (v_{src}, v_{dst}), which has no weight information yet.
- *Stage 2:* We choose a color for the vertex v_{src} if it has not been processed, and scan the table from $n{\cdot}v_{src}+1$ to $n{\cdot}v_{src}{+}n$. When the table value is empty, we mark the unit i as TRUE (i is a value from 1 to n), which means that the vertex v_{src} is assigned the ith color.
- *Stage 3:* If the value i is among 1 to $n-1$, we should make sure that the v_{dst} would not be the same color such that there are no adjacent vertices in these partitions. To achieve this purpose, we mark the position $n{\cdot}v_{dst}{+}i$ in the table by FALSE. If the value i equals to $n-1$, the vertex v_{src} must be put into the hybrid partition p_n, then we should not process vertex v_{dst}.



(a) Graph with three colored vertices V1, V2, V3.

	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
P_1	√	x	x	x	x			
P_2			√			x		x
P_3		√			x	x		x

(b) Confliction table with vertices V1, V3 and V2 processed.

	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
P_1	√	x	x	x	x	√	√	x
P_2			√	x	√	x	x	x
P_3		√		√	x	x	x	√

(c) Confliction table with all vertices processed.

(d) Final hybrid coloring results using only three colors/partitions.

Fig. 3. A concrete example with our coloring algorithm: (a) The example graph and three vertices have been colored. (b) V_3 and V_2 can not be divided into the same partition with V_1 as they are adjacent, so V_3 is set to the partition 2 and V_2 to partition 3. (c) The final result of the conflict table. (d) The final coloring result of our novel algorithm, and we color the graph with a limited number of colors and allow some neighbor vertices in the same partition, such as V_4, V_2, V_8.

By partitioning the graph by colors, we can update vertices of partitions from p_1 to p_{n-1} in parallel. There are no data conflicts inside each partition. For the last partition p_n, we process its vertices and edges sequentially or use GPU atomic operations.

We use the example shown in Fig. 3 to further illustrate our solution. In this example, we have a graph of eight vertices and divide the graph into three partitions, as shown in Fig. 3. The coloring algorithm loads the edges and processes associated vertices. Fig. 3a shows three vertices that are processed first and the order being processed is V_1, V_3, V_2 (which means that we do not need sorted *ids* of vertices). Fig. 3b shows the conflict table named *CT* after processing such three vertices. In the conflict table, vertex v will be assigned a *Processed* flag at row j if the vertex v is put in the partition j, or an *Unprocessed* flag if there is a conflict between vertex v and some other vertices. For this example, the size of *CT* is 3 × 8. Based on the table shown in Fig. 3b, we can know that vertex 1 is put in the partition p_1 as there is a *Processed* flag in *CT* [1,1], and obviously vertex V_3 and V_2 cannot be set to the same partition. After processing all of the edges, we can generate the conflict table eventually, as shown in Fig. 3c. We complete the coloring processing within O($|V|$+$|E|$) time complexity. Fig. 3d shows the result of our coloring algorithm. We use only three partitions, allowing some vertices (such as V_4, V_2, V_8 in this example) to be assigned different colors in the last hybrid partition.

3.3 Determining the Number of Colors/Partitions

The number of partitions must be set at the beginning of our coloring algorithm. With the consideration of the five-color theorem and the analysis of a real-world graph named DBLP (performance result shown in Fig. 7b), we find that it is usually viable to color a given graph under the consistency model with only five colors. Actually, the number of colors/partitions can be set to any viable number by the users/programmers. Although five colors may not be the best choice for Frog, we observe most of real-world graphs can get a good performance with only five colors: four colors contain about 80 percent of vertices and the last color includes the rest 20 percent of vertices. So we use the number 5 in our evaluation.

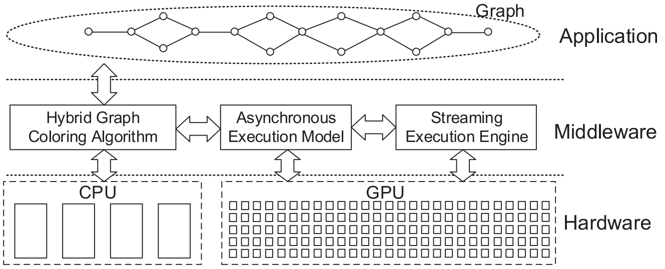

Graph
Application
Hybrid Graph Coloring Algorithm
Asynchronous Execution Model
Streaming Execution Engine
Middleware
CPU
GPU
Hardware

Fig. 4. System architecture.

When processing a large-scale graph, one of the partitions may contain too many vertices to be held in the GPU memory at a time. In order to address this issue, we split each color set into multiple pieces such that they can be put in the GPU memory. Moreover, the last one color set could be further colored for reaching the maximal parallelism especially for a large-scale graph, because its last partition may still have a large number of vertices and edges even though it is only 20 percent of the total amount. The further recoloring step on the last partition, however, may not work effectively for a median-size graph because the number of nodes/edges in the last partition is not that large.

4 Design Overview and Implementation

We design a light-weight asynchronous model in handling large-scale graphs based on our proposed hybrid graph coloring algorithm. This section describes the execution model and the details of our implementation.

4.1 Design Overview

The system architecture is shown in Fig. 4. Our design is focused on the middle-ware layer, which analyzes the applications and processes the related graphs using our optimized hybrid graph partitioning algorithm. It includes three parts: a hybrid graph coloring algorithm, an asynchronous execution model, and a streaming execution engine on GPUs. The first two parts are the most critical in our design.

We partition the graph based on the colors assigned to vertices, and process/update the vertices in parallel. Vertices with the same color are partitioned into the same chunk (also called *color-chunk*), which is to be processed in a separate *color-step*. A color-chunk contains the vertices which can be processed/updated without violating the sequential consistency. In the example shown in Fig. 5, the vertices 1, 6, 7 can be updated meanwhile because they have no adjacent edges, thus they can be put in a color-chunk and updated in parallel. Our algorithm ensures that no adjacent vertices are partitioned into the same color-chunk except for the last one. That is, the last color-chunk (and only this color-chunk) may contain some adjacent vertices, in order to reduce the number of colors for majority of vertices. In other words, the only possible synchronous processing occurs in the last hybrid *color-chunk*.

The asynchronous execution model aims to process the color-chunks generated by our coloring algorithm one by

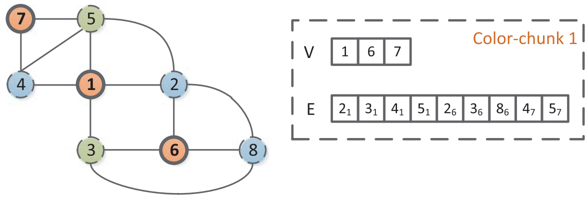


Fig. 5. An example of color-chunk.

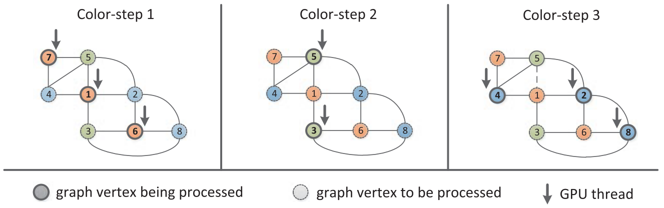


Fig. 6. Illustration of the hybrid coloring method using an example.

one, and also to guarantee the sequential consistency for each color-chunk (including the last hybrid color-chunk) meanwhile. For the last color-chunk that contains adjacent vertices and edges to process, there are two alternative solutions, either updating them sequentially or adopting atomic based operations. We find that if we use a sequential updating strategy, we cannot make full use of the powerful processing capacity provided by GPUs. Instead, the other strategy leads to the limited extra costs on GPU, due to the fact that the total number of vertices in the last hybrid color-chunk is always not very large (only about 20 percent or fewer vertices of the entire graph). Such a finding is confirmed in our experiments. In addition, our asynchronous execution model is also responsible for coordinating the message transmission among different chunks.

4.2 Asynchronous Execution

4.2.1 Graph Representation and Preprocessing

In the beginning of the processing work, our system checks if there exists some colored result to be processed in the data graph. If the answer is yes, the result can be used directly, which is one feature of our coloring algorithm. Otherwise, we need to process the data graph and divide vertices into several partitions based on the relaxed graph coloring algorithm.

After processing the data graph, we will get several color-chunks of vertices. Then we load graph edges based on the distribution of vertices and begin to partition the graph edges. Graph color-chunks are represented as Compressed Sparse Rows (CSR) in CPU memory. Furthermore, the array S which represents the local state of each vertex is necessary for our approach. In every color-step, update functions will access values of adjacent vertices without any changes. Thus, the capacity of array S must be larger than the number of partitioned vertices. We divide graphs into some partitions by the preprocessing/hybrid coloring algorithm, which also reduces random memory accesses inside each color-chunk.

4.2.2 Color-Step Iterative Asynchronous Execution

Our asynchronous approach processes graph partitions iteratively, as most of the graph algorithms follow iterative calculations. In our asynchronous approach, we use *color-step* (in analogy to the super-step in the BSP model) to describe the process of updating all the vertices in a single partition. As we divide the graph into n partitions, n color-steps should be processed. For each color-step, we conduct data transfers and GPU kernel function executions at the same time under the support of CUDA streaming technology, to reduce the overhead of PCIe data transfers.

Fig. 6 shows the computational abstraction based on a sample graph, which is processed with 3 colors/partitions. As we have already divided the graph into several partitions based on our hybrid graph coloring algorithm, vertices V_1, V_6, and V_7 can be simultaneously processed in the color-step 1 (shown in the left sub-figure), and the vertices V_3 and V_5 (shown in the middle sub-figure) can be processed in color-step 2 in parallel. As shown in the right sub-figure, vertices V_2, V_4, and V_8 of color-step 3 are assigned to the same color-chunk, which is a hybrid partition because V_2 and V_8 are adjacent vertices.

We group n color-steps into two categories: *P-step* and *S-step*. The first $n-1$ color-steps will be processed as the *P-step*s, which means vertices and edges of color-step can be updated *in parallel* without concerning the data conflicts; for the nth hybrid partition (the *S-step*), vertices and edges must be processed *sequentially* or using GPU atomic operations. The basic scheme is the simple kernel execution as per one color-step.

While processing graphs with the scale out of GPU memory, we only need to make sure that the scale of each *P-step* or *S-step* is not out of GPU memory. This can be guaranteed by the preprocessing stage, which means that we should not assign too many vertices to each color-chunk and may use more colors to partition out of GPU memory graphs.

4.2.3 Processing Hybrid Vertices Using Atomic Operations

To utilize the GPU resource, we combine several different colors together into the last partition, serving as the *S-step* to be processed on GPU. As mentioned previously, for most of graphs, there always exist some colors that consist of only a few vertices after the color partitioning. If we process them in a separate color-step and invoke one kernel execution, the worst case is that only a few or dozens of vertices would be updated by GPU stream processors.

For kernel execution of *S-step* to process the hybrid partition, we use atomic operations to guarantee the sequential consistency. Due to the limited number of vertices and edges to be updated serially, the overhead of using atomic operations is acceptable. The other workable solution is to process vertices and edges sequentially on CPU cores.

5 Performance Evaluation

In this section, we present experimental results in three different categories. First, we partition the graphs based on our improved coloring algorithm, to evaluate the efficiency of our hybrid approach on different colors. Second, we investigate the impact of the different numbers of partitions to the performance of our asynchronous method. Finally, we

TABLE 1
Properties of Graphs Used in Our Experiments

Datasets	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter
Vertices	735,322	986,286	1,134,890	1,965,206	2,394,385	41,652,229
Edges	5,158,012	6,707,236	2,987,624	2,766,607	5,021,410	1,468,365,167

present the execution performance under our asynchronous model with the hybrid coloring algorithm by comparing to five other related systems [2], [7], [8], [9], [10].

5.1 Experimental Setting

5.1.1 Experimental Environment

We conduct our experiments on a Kepler-based GPU, NVIDIA Telsa K20m with 6 GB main memory and 2,688 CUDA cores. We compile the CUDA programs with CUDA 7.0 using the *-arch=sm_35* flags. We directly reuse the source code of those engines from the authors.

5.1.2 Datasets, Testing Algorithms and Systems

Datasets. We evaluate six real-world graphs with different properties in our experiments, as shown in Table 1. Amazon, DBLP and Twitter are real-world graphs from [25], and others are from SNAP Datasets [26].

Testing Algorithms. We implement four common graph algorithms, namely PageRank, Breadth First Search (BFS), Single Source Shortest Path (SSSP) and Connected Component (CC). We mainly focus on the first two algorithms in the evaluation, because they are the most representative ones in the graph theory community.

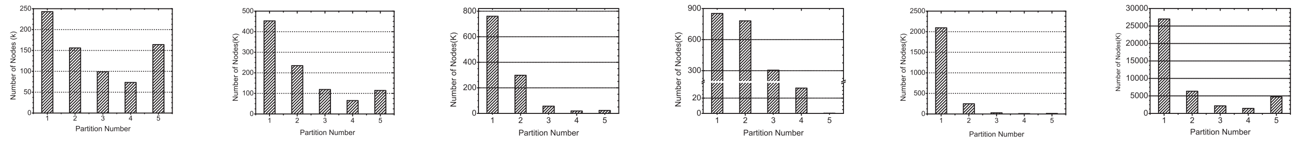
- *PageRank*. PageRank [27] has a strong demand for edge consistency model. While updating the rank value of vertex v, the rank values of all of its neighbor vertices that have outbound edges are also supposed to be updated. If implemented based on BSP model, rank values changed at current iteration only can be seen by other vertices at next super-step. For asynchronous approach execution, changes can be known by the function that updates other vertices which start from next color-step. This leads to a great convergence speed for PageRank.
- *BFS*. BFS is a widely used graph search algorithm which has a low computation-to-communication ratio. BFS mainly performs memory lookups rather than computations; hence its performance is more sensitive to memory access latency. The kernel implementation of BFS is based on the one that is used in Medusa [8]. This implementation of BFS explores all neighboring vertices from a starting vertex, in a level-by-level manner.
- *SSSP*. In the SSSP problem, we aim to find the shortest paths from a particular source vertex to all other vertices in the graph (if they exist). The traditional approach to SSSP problem is Dijkstra's method, which is also our choice. We assign random values between 1 and 64 to the edge values used in SSSP.
- *CC*. A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the other subgraphs.

Testing Systems. We evaluate five other GPU-based systems as compared with our systems. To analyze the effect of the hybrid/relaxed graph coloring algorithm, we add Frog-Native to compare with Frog. Frog treats the undirected graphs as a kind of special directed graphs to process, as also did by Cusha and Medusa.

- *Medusa*. Medusa [8] embraces an optimized runtime system with a simplified graph processing on GPUs. However it can only process graphs whose scales are smaller than that of device memory.
- *Totem*. Totem [28] is a hybrid system that partitions the graph and processes tasks using both the traditional CPU cores and GPU. There are three different graph partitioning strategies presented in Totem, HIGH-degree, LOW-degree and RAND-degree. HIGH-degree divides the graph such that the highest degree vertices are assigned to the CPU, while LOW-degree divides the graph with the lowest degree vertices being assigned to the CPU. For RAND-degree, vertices are randomly assigned to the CPU. Totem allows users to set the percentage of edges assigned to different devices. In our experiments, we load all graph edges to the GPU device to evaluate the performance.
- *Cusha*. Cusha [7] is a vertex-centric graph processing framework focusing on exploration of new graph representations named G-Shards and Concatenated Windows. We use the CW method, leading to the best performance of CuSha in our experiments.
- *MapGraph*. MapGraph [9] is a parallel graph programming framework based on Gather-Apply-Scatter (GAS) model. There are two different strategies adopted by MapGraph to improve the performance of Gather and Scatter phase. We use the default setting in MapGraph to choose optimized strategies in our experiments.
- *Gunrock*. Gunrock [10] is a high-performance graph processing library on GPU. The input parameters of algorithms we used in our evaluation are the same as that of its publication [10].
- *Frog-Native*. Frog-Native is a non-preprocessed synchronous graph processing system. The main difference between Frog and Frog-Native is that Frog-Native uses BSP model, while Frog uses a hybrid coloring model and asynchronous execution. In the computation phase of BSP model, Frog-Native uses atomics to ensure the data consistency and edge-centric programming model which is the same as Frog. Other implementations such as the realization of four graph algorithms are the same as Frog.

5.2 Hybrid/Relaxed Graph Coloring Algorithm

As mentioned previously, we color the graphs by only 5 colors, which is sufficient to fully utilize GPU resources. Our graph coloring algorithm guarantees that vertices in the first four partitions have no adjacent vertices. For the last partition, we combine a few different colors that are used in other ordinary heuristic coloring algorithms together.



(a) Amazon (b) DBLP (c) YouTube (d) RoatNet-CA (e) WikiTalk (f) Twitter

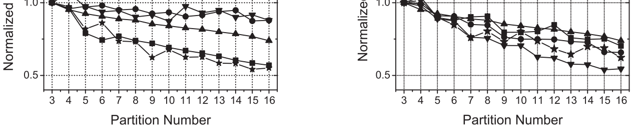
Fig. 7. Results of graph coloring and partitioning, using five partitions to divide graphs.

Fig. 7 shows the results of graph partitioning in numbers of vertices in the five partitions. For the first four partitions, all vertices in the same color-chunk share the same color, i.e., no adjacent vertices are in the same color-chunk. The number of vertices assigned to the first four partitions is nearly about 80 percent of the total number of vertices in the entire graph, or even more than this percent. For the last partition, there are only about 20 percent of vertices. For some graphs, such as the WikiTalk graph shown as Fig. 7e, they can be partitioned almost completely using only five colors. In addition, we observe that the first partition of most graphs always holds the most vertices which can take full advantage of GPU resource by having enough tasks to be processed at the same time.

Based on the above analysis, the hybrid scheme is suitable to partition the graphs, in that we can process majority of vertices with the satisfaction of sequential consistency and minority using atomic operations. More importantly, there are still enough vertices to process which can utilize the GPU resource. For graphs of different scales, however, some of the partitions may not consist of enough vertices. This is the reason why we need to analyze the effect of partitioning the graph using different numbers of colors. We conduct some experiments to study how the number of partitions affects the performance of our asynchronous approach.

Fig. 8 shows the performance of our asynchronous approach using different numbers of partitions execution running on BFS and PageRank. For graphs partitioning into different number, the performance improvement is remaining at around 0.5X - 1.0X. We set the performance using 3 partitions as the baseline.

For two real-world graphs, DBLP and WikiTalk, both of them need a lot of colors to complete the coloring work, 119 colors for graph DBLP and 81 colors for graph WikiTalk. When we add a partition (suppose the number of partitions is changed from k to k+1), only a few vertices and edges are put in the extra partition k, while most of vertices still stay in the last mix-color partition. The color-step invoked by these vertices obviously cannot utilize the GPU resource. For example, when we split the graph DBLP into six partitions, the additional partition only holds 5.89 percent of vertices from the previous hybrid partition using five partitions. The total processing time of this additional partition, however, is nearly 20 percent of the previous time, which leads to a lower processing capacity (only 5.89% / 20% = 29.5% of the previous processing). This means that when processing these few vertices on GPU, many processors are not utilized, which leads to significant resource idling.



(a) Performance on BFS (b) Performance on PageRank

Fig. 8. Effect of different partition numbers on the BFS and PageRank algorithm.

So far, we get different performance with different partition numbers being used. We always get a maximum performance gain using a moderate coloring and partitioning number, which happens while using 3 colors or 5 colors for real-world graphs.

5.3 Performance of Asynchronous Approach

Asynchronous computations can substantially accelerate the convergence of many algorithms in comparison with the BSP model. We process all the graphs using five partitions in the evaluation. The whole execution flow of graph processing mainly contains four stages: (1) disk I/O stage: reading graph data from disk; (2) data preprocessing stage (for example, partition the data if needed); (3) data transferring from CPU to GPU: also known as I/O overhead between CPU and GPU; (4) task execution stage: processing data on GPU. Stage 2 and 4 are two key steps we are focused on. In this section, we record the execution time in stage 4 (shown in Tables 2, 3 and 4) to analyze the performance of four algorithms, and record the time of other three steps (shown in Fig. 9) to mainly analyze the performance of preprocessing stages. Specifically, we analyze the reason why we get a good performance by presenting the number of iterations and atomic operations in the graph processing.

Table 2 presents the performance comparison between our asynchronous approach and five other systems by running BFS algorithm. Our algorithm has clear performance gains as shown in the table except compared with Gunrock [10]. Only except for the RoadNet-CA dataset, we get similar performance with Gunrock and better performance than MapGraph [9]. One of the reasons is that our system is particularly designed for the power-law graphs, while RoadNet-CA does not exhibit typical power-law feature. Another reason is that BFS algorithm cannot benefit from the asynchronous model, while Gunrock and MapGraph optimize the processing of BFS by using the frontier to reduce computation. As shown in Table 2, we obtain 6.55X performance improvement over Cusha [7].

We also get a high performance gain on PageRank, with an improvement of 4.32X over Medusa [8], as shown in Table 3. In particular, our asynchronous approach scales up to 55.15X on the algorithm PageRank when processing the graph WikiTalk. We run the PageRank algorithm when the damping factor is 0.85 and the precision is 0.005.

From the perspective of the overall system performance, our Frog also outperforms other state-of-the-art systems in

TABLE 2
Execution Time of BFS (in Milliseconds)

Datasets Systems	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter	Improvement
Frog-Native	9.882	7.631	13.286	223.606	24.143	null	–
Frog	10.587	6.980	**5.635**	176.856	**4.583**	**1,883.06**	–
Totem	39.265	38.220	33.143	254.798	72.225	null	1.44X–15.76X
Medusa	10.554	7.066	31.053	200.674	4.695	null	0.997X–5.51X
CuSha	34.321	31.856	19.805	347.314	30.009	null	1.96X–6.55X
MapGraph	10.467	11.285	9.549	82.112	17.996	null	0.46X–3.93X
Gunrock	**5.859**	**4.874**	7.246	**60.166**	5.675	null	0.34X–1.29X

Note: Null means that the system can not process such dataset on GPU, and the improvement is made based on Frog.

TABLE 3
Execution Time of PageRank (in Milliseconds)

Datasets Systems	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter	Improvement
Frog-Native	163.872	268.201	117.421	171.620	790.033	null	–
Frog	**34.723**	**50.813**	**35.655**	**20.370**	**75.130**	**144,893.98**	–
Totem	549.530	929.635	591.573	1,443.650	1,231.735	null	15.83X–70.87X
Medusa	149.916	313.145	503.283	165.758	4,143.684	null	4.32X–55.15X
CuSha	85.507	113.223	395.699	94.925	227.455	null	2.23X–11.10X
MapGraph	180.730	298.002	292.432	168.001	876.102	null	5.20X–11.66X
Gunrock	211.509	262.512	719.603	237.864	338.730	null	4.51X–20.18X

Note: Null means that the system can not process such dataset on GPU, and the improvement is made based on Frog.

TABLE 4
Execution Time of SSSP and CC (in Milliseconds)

Algorithm	Dataset	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter	Improvement
SSSP	Frog-Native	74.348	192.882	111.223	1411.780	241.231	null	–
	Frog	**34.320**	**47.461**	**13.042**	**280.390**	**24.412**	**65,708.65**	–
	Totem	54.890	53.926	30.393	567.612	91.923	null	1.14X–3.77X
	CuSha	86.038	86.023	34.438	1115.650	53.479	null	1.81X–3.98X
	MapGraph	72.541	78.820	29.779	785.858	107.598	null	1.66X–4.41X
	Gunrock	38.952	53.251	19.813	291.500	52.035	null	1.04X–2.13X
CC	Frog-Native	24.601	39.910	29.823	270.362	68.091	null	–
	Frog	**8.718**	**9.207**	**6.180**	247.850	**11.053**	**35,027.73**	–
	Totem	34.775	31.048	24.123	741.287	47.645	null	2.99X–4.31X
	CuSha	34.930	32.588	23.526	388.392	21.011	null	1.57X–4.01X
	MapGraph	88.608	78.963	56.893	1,834.050	114.630	null	7.40X–10.37X
	Gunrock	26.787	30.527	18.038	**31.362**	13.050	null	0.13X–3.32X

Note: Null means that the system can not process such dataset on GPU, and the improvement is made based on Frog.

most of cases. For example, we get a better performance than the graph processing framework CuSha [7] while running PageRank. As an asynchronous execution system, Frog exhibits a performance improvement from 1.96X to 6.55X over CuSha while running the BFS algorithm. Although CuSha partitions the graphs into different G-shards, which is similar to Frog, it generates too many partitions with a lot of duplicate values. In the PageRank case, Frog always outperforms Gunrock in any of the data sets.

In addition, Table 4 shows the performance improvement of Frog compared with CuSha, Gunrock and Totem on SSSP and CC algorithms. As for RoadNet-CA, Gunrock has higher convergence rate on CC algorithms because of its filter operator, while Frog does not make separate optimization for any one of algorithms. Overall, Frog gets a better performance over other system in most situation when comparing the execution time on GPU.

Frog also exhibits a better performance than preprocessed approach CuSha in the preprocessing stage, as shown in Fig. 9. As for the GPU-based systems listed in Fig. 9, only Cusha has the preprocessing stage before processing graph data on single-node GPU, in that Cusha adopts the shard concept to achieve high GPU utilization and it needs to take into account the size of the SM shared memory and generates hundreds of shards in general. In Fig. 9, we also observe that the preprocessing time of Frog always takes a very small portion of its total time of the three stages.

We record the numbers of iterations of all benchmarks, which confirms our asynchronous approach can accelerate the convergence with fewer iterations. Table 5 shows that we can reduce the number of iterations than other systems in most of the benchmarks, indicating much less overhead in the whole calculation process under our

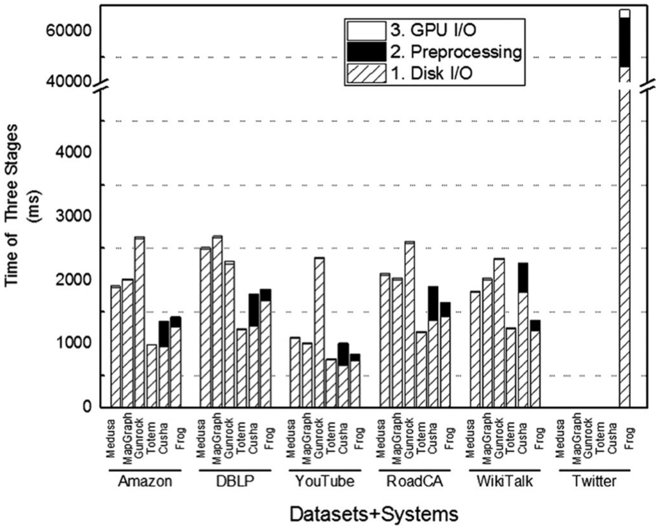


Fig. 9. Time of three stages before graph processing on GPU. Compared with the preprocessed approach Cusha, the time used in the preprocessing stage of Frog is just approximately 1/3 fraction of that in Cusha.

solution Frog. For algorithm CC, Gunrock has only two iterations for all kinds of datasets, which is the key reason why Gunrock exhibits better performance than Frog on CC in some cases.

We also record the numbers of atomic operations for all systems on GPU in the compute stage. Atomic operations on GPU are used to avoid race conditions and they simplify the coding for programmers. It is well-known that atomics can be used on GPU [24], but over-use of atomics may degrade the performance significantly. In fact, it is difficult to quantify the cost of an atomic operation on GPU accurately, because the cost of atomics in each system is related to many factors. As indicated by [24], some programming skills depending on different algorithms are leveraged to avoid atomics, and they are adopted by PageRank of Totem and CC of Gunrock to realize atomic-free design respectively. Table 6 shows that Frog has fewer atomics operations than other systems in most cases, especially in RoadNet-CA. Because only the last partition of data in Frog needs to use atomics and the number of atomics of Frog is closely related to the result of coloring partition.

To analyze the effect of the hybrid graph coloring algorithm in Frog, we conduct comparative experiment with Frog-Native. We observe that Frog significantly outperforms Frog-Native in almost all of our benchmarks, as it has fewer iterations and atomics than Frog-Native.

The reason our solution has a very high performance is two-fold. (1) The asynchronous model we adopt can converge faster than the synchronous model, because the updated value in the asynchronous model can be quickly accessed by other vertices. So, for the iterative graph algorithm, we have fewer iterations compared with other synchronous systems in GPU, which has been confirmed in our experiments (see Table 5). (2) We use hybrid coloring algorithm to implement an asynchronous execution model. Our strategy suffers from fewer atomics (see Table 6) in ensuring the data consistency on GPU as compared with the locking policy. Atomics may degrade performance significantly if they are used inappropriately, so we suffer less risk introduced by atomic operations.

TABLE 5
Comparison of Iteration Number on Frog and Competitive Systems

Algorithm	Dataset	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter
BFS	Cusha	27	13	10	553	6	null
	Totem	29	15	15	556	7	null
	MapGraph	30	15	23	557	7	null
	Gunrock	21	15	15	557	8	null
	Frog-Native	30	15	23	557	7	null
	Frog	29	14	22	556	6	14
Page Rank	Cusha	62	72	65	67	70	null
	Totem	126	165	13	143	107	null
	MapGraph	34	41	55	44	67	null
	Gunrock	46	47	50	26	50	null
	Frog-Native	52	51	15	31	33	null
	Frog	20	25	10	16	19	15
SSSP	Cusha	36	17	41	1097	6	null
	Totem	30	15	15	556	7	null
	MapGraph	30	15	23	557	7	null
	Gunrock	46	57	29	694	12	null
	Frog-Native	28	14	18	555	8	null
	Frog	17	8	14	279	4	7
CC	Cusha	27	13	18	551	2	null
	Totem	20	12	10	555	5	null
	MapGraph	20	15	14	555	8	null
	Gunrock	2	2	2	2	2	null
	Frog-Native	35	32	26	561	20	null
	Frog	9	7	8	252	5	6

Note: Null means that the system can not process such dataset on GPU, and the improvement is made based on Frog.

6 Related Work

6.1 GPU-Accelerated Graph Processing

Much research has been conducted on the efficient processing of large graphs due to the increasing number of graph-based applications. Some researchers used GPU to accelerate graph algorithms [23], [29], because GPUs have significantly higher memory access bandwidth and much higher parallelism. Hong et al. [19] present a BFS implementation on a GPU-accelerated platform and divide the processing into two phases. The phase on GPU starts as soon as enough parallelism is exposed to utilize the GPU memory bandwidth more efficiently.

Medusa [8] and MapGraph [9] both are the programming frameworks which provide a set of APIs for developers to implement their applications simply on GPUs. Medusa enables developers to leverage the capabilities of GPUs. The system loads the whole graph into GPU memory and automatically executes user-defined operations in parallel. MapGraph adopts dynamic scheduling strategy and two-phase decomposition strategy to decrease the thread-divergence penalty and balance the workload of threads. Totem [2] is a hybrid system that partitions the graph and processes tasks using both the traditional CPU cores and the GPU. By uploading a suitable number of vertices onto the device, Totem is able to process graphs that not fit the GPU memory size. CuSha [7] is a framework which makes it easy for users to define vertex-centric algorithms for processing large graphs on GPU. Gunrock [10] is a high-level graph library on the GPU which provides a novel data-center abstraction and several GPU-specific optimization strategies.

6.2 Graph Partitioning

While processing large-scale graphs, partitioning is a major step which affects the performance of graph processing. A

TABLE 6
Comparison of the Number of Atomic Operations on Frog and Competitive Systems

Algorithm	Dataset	Amazon	DBLP	YouTube	RoadNet-CA	WikiTalk	Twitter	Operation Ratio
BFS	Cusha	1.0E+8	6.6E+7	2.5E+7	1.6E+9	2.3E+7	null	1.4E+1X–5.7E+6X
	Totem	1.5E+6	1.6E+6	7.4E+6	4.2E+6	4.7E+6	null	2.8E+0X–1.5E+4X
	MapGraph	7.7E+5	9.2E+5	4.5E+5	1.9E+6	1.3E+7	null	1.3E+0X–7.0E+3X
	Gunrock	1.3E+6	8.6E+5	6.8E+5	1.9E+4	2.6E+8	null	2.0E+0X–1.6E+2X
	Frog-Native	1.3E+6	1.5E+06	1.1E+6	2.3E+6	3.1E+6	null	1.9E+0X–8.3E+3X
	Frog	**1.8E+5**	**4.0E+5**	**3.4E+5**	**2.8E+2**	**1.6E+6**	**1.9E+7**	–
PageRank	Cusha	3.2E+8	4.1E+8	7.5E+7	3.9E+8	3.0E+8	null	3.2E+2X–9.5E+4X
	Totem	**0.0E+0**	**0.0E+0**	**0.0E+0**	**0.0E+0**	**0.0E+0**	null	–
	MapGraph	7.7E+5	9.2E+5	1.5E+6	2.0E+6	1.8E+6	null	3.2E+0X–4.9E+2X
	Gunrock	3.7E+6	4.0E+6	3.1E+6	5.3E+6	4.8E+6	null	1.3E+1X–1.3E+3X
	Frog-Native	2.9E+7	2.6E+7	7.7E+6	2.1E+7	7.1E+7	null	3.3E+1X–5.2E+3X
	Frog	**2.4E+5**	**1.7E+5**	**2.3E+5**	**4.0E+3**	**2.0E+5**	**5.5E+7**	–
SSSP	Cusha	4.3E+8	3.3E+8	8.4E+7	1.9E+9	6.7E+7	null	1.3E+2X–1.2E+5X
	Totem	9.7E+7	1.2E+8	4.7E+7	9.5E+8	1.7E+8	null	3.0E+2X–6.1E+4X
	MapGraph	1.8E+7	1.8E+7	6.9E+6	1.8E+8	2.4E+7	null	4.4E+1X–1.2E+4X
	Gunrock	5.9E+6	1.1E+6	9.5E+5	2.5E+5	1.1E+9	null	3.0E+0X–2.1E+3X
	Frog-Native	7.5E+6	6.3E+6	2.7E+6	6.2E+7	6.8E+6	null	1.3E+1X–4.0E+3X
	Frog	**2.4E+5**	**3.5E+5**	**1.6E+5**	**1.6E+4**	**5.2E+5**	**5.8E+7**	–
CC	Cusha	9.9E+7	6.6E+7	2.5E+7	1.6E+9	2.3E+7	null	1.3E+1X–1.2E+5X
	Totem	7.1E+7	7.5E+7	4.3E+7	1.9E+9	1.4E+8	null	6.1E+1X–1.4E+5X
	MapGraph	6.5E+7	6.1E+7	2.2E+7	7.6E+8	3.5E+7	null	2.0E+1X–5.7E+4X
	Gunrock	**0.0E+0**	**0.0E+0**	**0.0E+0**	**0.0E+0**	**0.0E+0**	null	–
	Frog-Native	9.5E+7	7.6E+7	2.9E+7	2.8E+8	4.9E+7	null	2.8E+1X–2.1E+4X
	Frog	**1.2E+6**	**8.6E+5**	**4.4E+5**	**1.3E+4**	**1.7E+6**	**3.8E+6**	–

Note: Null means that the system can not process such dataset on GPU, and Operation Ratio is referred to as the ratio of the atomic operation of other systems over Frog.

lot of work has been done on graph partitioning. PowerGraph [22] is a distributed graph system that overcomes the workload imbalance problem with vertex-based partitioning with replication. GPS [30] extends Pregel's API and supports dynamic repartitioning based on the outgoing communication. Tian et al. [31] introduce a graph-centric programming model that opens up the partition structure to users and allows information to flow freely inside a graph partition.

Both Medusa [8] and Hong et al. [19] assume that GPU memory can hold the entire graph. However, this is not true when processing large graphs. The benefit of GPU execution is limited by GPU's relatively small memory capacity. This is a major limitation of large-scale graph processing on GPUs. Totem [2] is introduced as a hybrid system that partitions the graph and processes tasks using both the traditional CPU cores and the GPU. Totem partitions the graph by placing the high-degree vertices in one type of processor and the low-degree ones in the other type, or placing random-degree vertices on two types of processor. However, this method does not adapt to the asynchronous execution because of the data dependent computation on graphs.

6.3 Synchronous/Asynchronous Execution Models

Pregel [4] is a vertex-centric framework based on the BSP model, which uses synchronous super-steps for computation and communication. Mizan [32] uses fine-grained vertex migration to achieve the workload balance across Pregel super-steps. Similar to systems such as Pregel, X-Stream [33] presents an edge-centric graph processing using streaming partitions for scale-up graph processing on a single shared-memory machine, which takes the cost of random access into the set of vertices instead of the set of edges with much larger scale. By exploiting the high level semantics of abstract data types, the Galois [34] system ensures that any parallel execution must be faithful to the sequential semantics and is able to allow concurrent accesses and updates to shared objects. Ligra [35] presents a simple shared memory abstraction for vertex algorithms which is particularly good for problems similar to graph traversal, which is motivated in part by Beamer et. al.s recent work on a very fast BFS for shared memory machines [36].

In contrast to synchronous execution models, which update all values simultaneously by using values from previous time step, asynchronous systems update them using the most recent values. In the paper [5], it demonstrates that how asynchronous computation can substantially accelerate the convergence of some algorithms like PageRank. GraphLab [15] and PowerGraph [22] present the framework based on the asynchronous computational model to achieve better convergence than the synchronous implementation. To exploit asynchronous parallelism in iterative algorithms on distributed shared memory, ASPIRE [37] uses a relaxed memory consistency model and cache consistency protocol, simultaneously maximizing the avoidance of long latency communication operations and minimizing the adverse impact of stale values on convergence.

7 Conclusion

GPUs have recently been adopted to accelerate various graph processing algorithms. Many existing frameworks and processing engines are based on the BSP model.

Unfortunately, the cost of synchronization becomes serious because of the increasing graph scale.

We present a light-weight asynchronous framework called Frog on GPU. We partition the graph based on a new efficient graph coloring algorithm to ensure that no adjacent vertices are divided into the same color-chunk for a large majority of vertices. Vertices of each partition, except the last hybrid partition, satisfy sequential consistency and can be updated in parallel without modifying the data in adjacent vertices. Some key findings based on our experiments are summarized below.

- We present a novel and relaxed graph coloring algorithm, which divides majority of vertices (about 80 percent of the entire graph) into a moderate number of partitions without adjacent vertices.
- We present an asynchronous framework based on our relaxed coloring algorithm. The scheduling scheme is one *color-chunk* for one kernel execution. Thus we can process most of vertices of *color-chunk* without adjacent vertices in parallel. By the experimental results, it can be seen that our system Frog exhibits better performance than the other three well-known GPU-based graph processing systems (Cusha, Medusa and Totem) on all cases. Gunrock and MapGraph exhibits better performance than our system on some cases. Specially, For CC, only the preprocessed system Gunrock performs better on the dataset RoadNet-CA than our system. For BFS, our system exhibits comparable performance with Gunrock on datasets except for Amazon, DBLP and RoadNet-CA, and outperforms the other four GPU-based systems on almost all datasets. The only exception is the case of RoadNet-CA, where MapGraph exhibits better performance than our system.
- We also present a light-weight streaming execution engine for handling the large scale graph on GPU, such as twitter-2010 graph with 1.468 billion edges. We get a better performance than Totem and CuSha, while the other systems like Medusa is unable to run these graphs in such a large scale.

Acknowledgments

The work is supported by the International Science and Technology Cooperation Program of China (No. 2015DFE12860), NSFC (No. 61370104, 61433019, U1435217), the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032), and the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

References

[1] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *Proc. ACM Symp. Cloud Comput.*, 2012, pp. 1–13.

[2] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proc. IEEE/ACM Int. Conf. Parallel Archit. Compilation Techn. Cite Publication*, 2012, pp. 345–354.

[3] C. W. Ou and S. Ranka, "Parallel incremental graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 884–896, Aug. 1997.

[4] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[5] Y. Low, et al., "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.

[6] L. Wang, J. L. Xue, and X. J. Yang, "Acyclic orientation graph coloring for software-managed memory allocation," in *Science China Information Sciences*. Beijing, China: Science China, 2014, pp. 1–18.

[7] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 239–252.

[8] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.

[9] Z. Fu, M. Personick, and B. Thompson, "MapGraph: A high level API for fast development of high performance graph analytics on GPUs," in *Proc. Workshop GRAph Data Manage. Experiences Syst.*, 2014, pp. 1–6.

[10] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, pp. 265–266.

[11] GTgraph, 2010. [Online]. Available: http://www.cse.psu.edu/madduri/software/GTgraph/index.html

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT : A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

[13] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[14] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," *ACM SIGARCH Comput. Archit. News*, vol. 35, pp. 278–289, 2007.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *UAI*, arXiv: 1006.4990, 2010.

[16] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. HLipasti, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *Proc. 34th ACM/IEEE Int. Symp. Microarchitecture*, 2001, pp. 328–337.

[17] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. Conf. Innovative Data Syst. Res.*, 2013, pp. 3–6.

[18] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *Proc. VLDB Endowment*, vol. 6, pp. 2014–2025, 2013.

[19] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 77–78.

[20] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Oper. Syst. Des. Implementation*, 2012, pp. 31–46.

[21] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Englewood Cliffs, NJ, USA: Prentice Hall, 1989.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Conf. Oper. Syst. Des. Implementation*, 2013, pp. 17–30.

[23] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 117–128.

[24] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," in *Proc. 6th Workshop General Purpose Processor Using Graph. Process. Units*, 2013, pp. 96–107.

[25] Laboratory for Web Algorithmics (LAW), Datasets, 2013. [Online]. Available: http://law.di.unimi.it/datasets.php

[26] Stanford Large Network Dataset Collection, Datasets, 2014. [Online]. Available: http://snap.stanford.edu/data/

[27] S Markus, The PageRank Algorithm, 2003. [Online]. Available: http://pr.efactory.de/e-pagerank-algorithm.shtml

[28] A. Gharaibeh, L. Beltrao Costa , E. Santos-Neto , and M. Ripeanu, "On graphs, GPUs, and blind dating: A workload to processor matchmaking quest," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 851–862.

[29] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. 14th Int. Conf. High Perform. Comput.*, 2007, pp. 197–208.

[30] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. Int. Conf. Sci. Statist. Database Manage.*, 2013, pp. 1–12.

[31] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "Think like a vertex" to "Think like a graph"," *Proc. VLDB Endowment*, vol. 7, pp. 193–204 , 2013.

[32] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 472–488.

[34] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proc. 28th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2007, pp. 211–222.

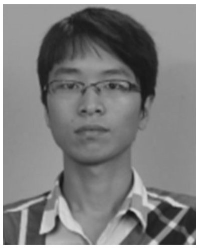
[35] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 135–146.

[36] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 10–16.

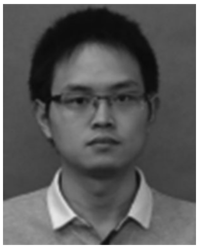
[37] K. Vora, S. C. Koduru, and R. Gupta, "ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2014, pp. 861–878.

Xuanhua Shi is a professor in the Big Data Technology and System Lab/Service Computing Technology and System Lab, Huazhong University of Science and Technology, in China. His current research interests focus on the cloud computing and big data processing. He published more than 70 peer-reviewed publications, received research support from a variety of governmental and industrial organizations, such as the National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union, and so on. He is a member of the IEEE.

Xuan Luo is working toward the master's degree in the Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL), Huazhong University of Science and Technology, in China. She is now doing some research on graph processing on GPU.



Junling Liang is working toward the master's degree in the Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL), Huazhong University of Science and Technology, in China. He is now doing some research on graph processing on GPU.



Peng Zhao is working toward the master's degree in the Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL), Huazhong University of Science and Technology, in China. He is now doing some research on graph processing on GPU.

Sheng Di is currently a postdoctoral researcher in the Argonne National Laboratory. His research interest involves resilience on high-performance computing and broad research topics on cloud computing. He is working on multiple HPC projects, such as detection of silent data corruption, characterization of failures and faults for HPC systems, and optimization of mutlilevel checkpoint models. He is the author of 12 papers published in international journals and 30 papers published at international conferences. He served as a programming committee member 14 times for different conferences, and served as an external conference/journal reviewer more than 50 times. He is a member of the IEEE.

Bingsheng He is an associate professor in the Department of Computer Science, National University of Singapore, Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems. Since 2010, he has (co-)chaired a number of international conferences and workshops, including IEEE CloudCom 2014/2015 and HardBD2016. He has served on the editorial board of international journals, including the *IEEE Transactions on Cloud Computing* (*IEEE TCC*) and the *IEEE Transactions on Parallel and Distributed Systems* (*IEEE TPDS*). He is a member of the IEEE.

Hai Jin is a Cheung Kung Scholars chair professor of computer science and engineering with the Huazhong University of Science and Technology (HUST), in China. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System. He is a senior member of the IEEE.