Frog: Asynchronous Graph Processing on GPU with Hybrid Coloring Model

Xuanhua Shi¹, Junling Liang¹, Xuan Luo¹, Sheng Di², Bingsheng He³, Lu Lu¹, and Hai Jin¹

¹ Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
² Argonne National Laboratory, USA
³ School of Computer Engineering, Nanyang Technological University, Singapore
¹ {xhshi, junlingliang, luoxuan, llu, hjin}@hust.edu.cn, disheng222@gmail.com, bshe@ntu.edu.sg

Abstract—GPUs have been widely 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 the atomic operations, leading to significant penalties/overheads when implemented on GPUs. To this end, 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 hybrid coloring model. The fundamental idea is based on Pareto principle (or 80-20 rule) about coloring algorithms as we observed through masses of real graph coloring cases. We find that majority of vertices (about 80%) 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 will separate the processing of the vertices based on the distribution of colors. In this work, we mainly answer the three questions: (1) how to partition the vertices in a spare graph with maximized parallelism, (2) how to process large-scale graphs which are out of GPU memory, and (3) how to reduce the overhead of data transfers on PCI-E while processing each partition. Experiments based on real-world data show that our asynchronous GPU graph processing engine outperforms other state-of-the-art approaches by 4.2X–29.7X.

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 very significant to develop a lightweight approach that can process large graphs with millions/billions of vertices/edges very efficiently.

GPUs are often used to accelerate the graph processing algorithms, not only because of the massive computation power, but also due to much higher memory bandwidth and lower latency. Still, there are some serious issues in the existing solutions in 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 [5], [28], [6] 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 Medusa [36], Totem [6], CuSha [16]) are designed based on the synchronous processing model - Bulk Synchronous Parallel (BSP) model [23]. The BSP model divides processing 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 the synchronous model, there are some asynchronous models that have been proved more efficient in processing graphs [22], but they are not very suitable for parallel graph processing on GPU. In the asynchronous model, 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 [22] and GraphChi [18]) 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 work¹, we design a lock-free parallel graph pro-

¹. Available at https://github.com/AndrewStallman/Frog.git
cessing 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 simultaneously in parallel, also guaranteeing the sequential consistency of the parallel execution.

Graph coloring is a classic problem [20], based on which one can process the graph in parallel and meanwhile satisfy the edge consistency model [22]. The key challenge is that optimizing graph coloring is an NP-complete problem, which means that it is impossible to find the optimal solution with polynomial time unless P=NP. As for the large graph with billions of vertices, existing graph coloring heuristics are, in general, not viable either. This is because coloring graphs with billions of vertices may generate 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%) are colored with only a small number of colors (about 20% or less), while only a small amount of vertices (about 20%) are assigned with a large number of different colors (about 80%). Based on such a finding, our solution will process the vertices based on their coloring distributions. In particular, we process 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% of vertices that are assigned different colors?
• How to reduce the overhead of data transfers on PCI-E while processing each partition?

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

1) An efficient hybrid graph coloring algorithm: We propose an efficient 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: After partitioning the graph, our execution engine scans the input graph color by color, updating all the vertices with the same color in parallel (one color as per kernel execution). Since the 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: While processing each partition, we overlap the data transfers and GPU kernel function executions, so as to minimize the overhead of PCI-E data transfers.

We evaluate our hybrid graph coloring model by comparing to four other state-of-the-art related systems. Experiments show that our solution obtains the performance gains by 3.7X–29.1X improvement over CuSha [16] on BFS algorithm, and 4.2X–29.7X over the related work Medusa [36] on PageRank algorithm. The performance under our solution also exhibits better than Totem [6] on both BFS and PageRank algorithm.

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 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

Traditional GPU can only handle 3D image rendering task, and CUDA products launched by NVIDIA reverse the situation. Through the effective use of its hardware resources, GPU can be used for general computing now. Typical general-purpose graphics units consist of many computation units called Stream Multiprocessors (SM). Each SM mainly contains one or more instruction units, multiple ALUs and a shared memory. These ALUs in a SM share the same instruction and threads mapped on these ALUs execute on different data, which means that the computing model of GPU is SIMD (Single Instruction Multiple Data). Moreover, for Nvidia’s GPUs, each group of threads executing the same instruction is called a warp, which contains 32 threads usually. And GPUs have significantly higher memory access bandwidth. No matter sequential or random memory access, it has a 20X to 30X speedup over CPUs.

Today’s GPUs have high computation power with massive multi-threads and become popular in graph processing due to the NVIDIA CUDA framework, and have been used widely to accelerate graph processing. While as a power computing processor, GPU has some excusable limitations. Firstly, GPUs have much less global memory. Also unfortunately, the PCI-E express bus, as the bridge of data transfer between CPU and GPU memory, can become a bottleneck because of the limited bandwidth. Usually, PCI-E transfers consume a large fraction (30% to 80%) of total execution time dedicated to CPU-GPU memory transfer and GPU kernel execution. We must have an efficient method
to enforce data transfers between these two different devices. Secondly, some general operations used on the CPU-based systems are very costly for GPU-based processing. Lock operations and atomics, for example, would be very expensive on massively parallel GPUs and often result in the critical path in a program. This is one of reasons why we need an efficient coloring based execution instead of using locks on GPUs. Thirdly, 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 [36], [6].

Our system Frog is aimed at solving the above problems. Considering the limited global memory, we divide the graph data by our coloring algorithm. After coloring, even for the large scale graph, memory can hold the graph data of each partition. In order to reduce the overhead of locking mechanism, our coloring algorithm ensures the execution of each partition. Our system also provides APIs for the developer to make the programming easier. So the developers can deal with graph processing without caring about the details of GPU programming model.

Not only we consider the GPU existing limitations, but also, through many experiments, we find two other important characteristics of GPU processors that should be paid attention to. We conduct these experiments using synthetic graphs generated by GTgraph [8], which can generate graphs with power-law degree distributions (or Pareto principle) and small-world characteristics based on Recursive MATrix (RMAT) model [4]. Each R-MAT graph in our experiments has $n$ vertices and $10+n$ edges. Our system is mostly concerned about the power-law graphs, which have been widely studied recently.

One characteristic is that the execution performance of GPU is related with the number of vertices to be processed. In order to keep a high execution efficiency, we should make sure there are enough vertices to be updated by a large number of threads (such as thousands of threads) when processing large-scale graphs. If the parallelism is not high enough, it would cost 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. Figure 1 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 K20x GPUs, in order to obtain a good performance.

Another characteristic is that the overhead of scheduling between different kernels is small. We adopt NVIDIA CUDA as our development platform, which is also a software environment for parallel computing on GPU. Parallel portions of the algorithm are executed on GPU as kernels. CUDA uses 1000 threads which can execute the same kernel to achieve performance gains, and the kernel can be launched very quickly on GPU. Figure 2 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 limited even we partition the graph and make kernel calls for each partition.

![Fig. 1: GPU execution time with different scale R-MAT graphs. It takes almost the same time while processing graphs with 10 vertices and 10k vertices, which means this scale of graphs can not utilize the GPU resources.](image1)

![Fig. 2: Execution time with different number of GPU kernels. 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.](image2)

2.2 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 Lampord, 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.” [19]. 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 [25], [3], [21].

There are three data consistency models [21] that can be used in graph processing, to guarantee the sequential consistency. Each one of them can make sure the correct results when the parallel algorithm performs the 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 $V_i$, 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 cannot read its adjacent vertices. The edge consistency model ensures each update function has 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.

Appropriate selection of the consistency model directly impacts 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, as we can get a higher parallelism than the full consistency model and pay less cost to ensure 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 a lower iteration speed.

### 2.3 Motivation of this work

For the purpose of easy programming and system deployment, the synchronous execution model is particularly suitable for 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 straightforward 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 research [22], [34], [35].

- **Non-negligible Synchronization Overhead**: Synchronous model is fairly suitable to process the situation 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 [13]. 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 Coordinations**: Many algorithms may not converge in the synchronous model, 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. While the other side, these algorithms can converge under the asynchronous model if we ensure the appropriate consistency at the same time.

- **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.

In comparison to the synchronous model, asynchronous model is rather difficult to design and manage, because 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 [22] and sequential execution [18]. Locking or atomic operations, however, would be very expensive on massively parallel GPUs [27], [26], thus neither locking protocol nor sequentially updating is a viable solution.

In comparison to the locking protocol and atomic operations, graph coloring method would be very efficient since it is a kind of asynchronous model which can process the vertices in parallel and guarantee the serializability meanwhile. The basic 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**, which is defined by GraphLab [21]. 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 Figure 3). This may result in a large number of small color groups each with only a few vertices to update, which finally significantly limits the parallelism.

Based on a lot of experiments (shown as the Figure 9 in the evaluation section), we observe a typical Pareto principle in the coloring process, i.e., a large majority of vertices are colored with only a few colors (such as four colors to the graph LiveJournal). This inspires us to improve the parallelism by leveraging the coloring distribution. On the other hand, optimizing the strict graph coloring is an NP-complete problem, which means that it is extremely difficult to find the optimal polynomial solution. Hence, we designed a relaxed graph coloring algorithm to color the graph based on the Pareto principle. Based on the relaxed coloring algorithm, we can further develop our
posed to be taken into account by coloring based graph algorithms relaxed and quickly. Last partition) if necessary, so as to complete our coloring number of colors and make a relaxed coloring. We combine source. Accordingly, in our coloring algorithm, we limit the only a few vertices, which will under-utilize the GPU re-coloring work. Most of the colors, however, actually involve coloring algorithms use a lot of colors to complete the same color. For large-scale graphs, general heuristic graph partitioning Strategies

3.1 Required Features for Coloring Based Graph Partitioning Strategies

Graph coloring is a classic algorithm that assigns a color to each vertex such that no adjacent vertices share the same color. For large-scale graphs, general heuristic graph coloring algorithms use a lot of colors to complete the coloring work. Most of the colors, however, actually involve only a few vertices, which will under-utilize the GPU resource. Accordingly, in our coloring algorithm, we limit the number of colors and make a relaxed coloring. We combine some adjacent vertices into the same hybrid partition (the last partition) if necessary, so as to complete our coloring algorithm relaxed and quickly.

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

- **Low time complexity:** it will be more and more costly to process a graph as its size grows. As the partitioning is just the preliminary step, that is, 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 Partitioning Strategy: Being partitioned by colors

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 \( p_1, p_2, p_3, \ldots, p_n \). For the first \( n-1 \) partitions \( (p_1, p_2, \ldots, p_{n-1}) \), 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 the 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 majority of adjacent vertices are assigned different colors. That is, our solution allows a few pairs of adjacent vertices (about 20% of vertices) to be assigned the same colors, 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 \( n \)th 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 exhibits 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 changed to a certain extent, 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 systematically present our coloring method, and then further 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 \(n-v_{src}+1\) to \(n-v_{src}+n\). 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 \(i\)th 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 \(n-v_{dst}+i\) 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 \(v_{dst}\).

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.

The pseudo-code of our hybrid coloring algorithm is presented in Algorithm 1. There are three input parameters, which are \(g\) (the graph to be processed), \(n\) (the number of colors/partitions), \(n_v\) (the number of vertices of input graph \(g\)). Then we initialize a table with \(n\) rows and \(n_v\) columns to mark the state of each vertex. **Partition array** records the detailed partition number of each vertex. Another auxiliary array is called **processed array**, which is used to record whether the vertex is processed (lines 2–4). Then we process a vertex \(v\) from graph \(g\) and the adjacent-vertex list containing all of its adjacent vertices. For the vertices which have been processed in the list, we mark the corresponding position with the **Unprocessed** flag (lines 14–19). Then we select the suitable position and assign the vertex \(v\) into this partition. In the latest version of our coloring algorithm, we choose the first empty position \(p\) as the suitable assigned partition (lines 20–26). For the vertices which have not been processed in the list, we mark all the position \(p\) of each vertex with the **Unprocessed** flag.

We use the example shown in Figure 4 to further illustrate our solution. In this example, we have a graph of eight vertices and divide the graph into three partitions, shown as Figure 4. The coloring algorithm loads the edges and processes associated vertices. Figure 4 (a) 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. Figure 4 (b) 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 \(3 \times 8\). Based on the table shown in Figure 4 (b), we can know that vertex \(v_1\) 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 Figure 4 (c). We complete the coloring processing within \(O(|V|+|E|)\) time complexity. Figure 4 (d) shows the result of our coloring algorithm. We use only three partitions, allowing some vertices (such as \(V_1, V_2, V_3\) in this example) to be assigned different colors in the last hybrid partition.

### Algorithm 1 Hybrid Coloring Method

**Input**: Graph \(g\); Number of partitions \(n\); Number of vertices \(n_v\)

1: \(\text{table}[n_v * n] \leftarrow 0\)  \hspace{1cm} \text{\(\triangleright\) Initialization phase}
2: \(\text{partition}[n_v] \leftarrow 0\)
3: \(\text{processed}[n_v] \leftarrow 0\)

6: \(\text{\(\triangleright\) Load edges from graph and process vertices}
7: while \(g\) is not NULL do
8: \(\text{PROCESSINGVERTEX}(vertex\ v, \text{list of outcoming edges})\)
9: end while
10: 11: \(\text{function } \text{PROCESSINGVERTEX}(vertex\ v, \text{list of outcoming edges})\)
12: \(\text{start} \leftarrow \text{starting position of vertex } v \text{ at the table}\)
13: \(\text{len} \leftarrow \text{length of the list of outcoming edges}\)
14: for each \(i < \text{len} \) do
15: \(\text{if } \text{processed}([\text{list}[i]]) == 1 \text{ then}\)
16: \(\text{pos} \leftarrow \text{partition}([\text{list}[i + 1]])\)
17: \(\text{table}[[\text{start}] + \text{pos}] \leftarrow -1\)
18: end if
19: end for
20: for each \(1 < n \) do
21: \(\text{if } \text{table}[[\text{start} + i]] == 0 \text{ then}\)
22: \(\text{table}[[\text{start} + i]] \leftarrow 1\)
23: \(\text{partition}([v]) \leftarrow i\)
24: \(\text{processed}([v]) \leftarrow 1\)
25: end if
26: end for
27: for each \(i < \text{len} \) do
28: \(\text{if } \text{processed}([\text{list}[i]]) == 0 \text{ then}\)
29: \(\text{pos} \leftarrow \text{list}[i] * n\)
30: \(\text{table}[[\text{pos} + \text{partition}([v])] \leftarrow -1\)
31: end if
32: end for
33: end function

### 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 large-scale graph named LiveJournal (performance result shown in Figure 9d), we find that it is usually viable to color a given graph under the consistency model with only five colors. In particular, we observe nearly 80% of the vertices in a large-scale graph are always able to be divided into the partitions without adjacent vertices. Actually, the number of colors/partitions can be set to any viable number. We use the number 5 to partition and evaluate most of our data graphs, as four colors can contain about 80% of vertices and the last color combine the rest 20%.

For a large-scale graph, one of the partitions may contain too many vertices such that the GPU memory is still not large enough to hold them at a time. For the first four partitions, we can just divide them into subsets based on the memory size, because each of them shares the same color. For the last hybrid partition, we re-process the subgraph by coloring these vertices again. We process the subgraph of the hybrid partition as a new graph with our relaxed coloring algorithm to guarantee that each partition generated by our coloring algorithm must fit the GPU memory.

### 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.
Fig. 4: A concrete example with our coloring algorithm: (a) shows the example graph and three vertices have been colored; (b) shows that $V_3$ and $V_2$ cannot 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) shows the final result of the conflict table; (d) shows the final coloring result of our novel algorithm, and we color the graph with limited number of colors and allow some neighbor vertices in the same partition, such as $V_4$, $V_2$, $V_5$.

4.1 Design Overview

The system architecture is shown in Figure 5. Our design is focused on the middleware 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 most critical in our design.

![Graph](image)

**Graph**

**Application**

**Hybrid Graph Coloring Algorithm**

**Asynchronous Execution Model**

**Streaming Execution Engine**

**Middleware**

**Hardware**

![Diagram](image)

Fig. 5: System Architecture. Frog includes three parts: a hybrid graph coloring algorithm, an asynchronous execution model, and a streaming execution engine on GPUs.

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 Figure 6, 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 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 as GraphChi [18] did, 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% 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 from the data graph to
be processed. If the answer is yes, the result can be used directly, which is one of features of our coloring algorithm. Otherwise, we need to process the data graph and divide vertices into serval 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 number of partitions by the preprocessing/hybrid coloring algorithm, which also reduces random memory accesses inside each color-chunk.

In summary, in the pre-processing phase before switching to the GPU processors, we need to prepare three types of data for each color-step, as listed below.

1. Vertex Array ($V$) contains vertices to be updated by GPU kernel functions;
2. Edge Array ($E$) contains all outgoing edges of vertex $v$ ($v \in V$), or incoming edges according to the updating algorithm;
3. State Array ($S$) stores values of vertices associated with edges $e$ ($e \in E$), not only vertices $v$ ($v \in V$), but also their adjacent vertices of $v$. Such a design makes some contents of $S$ able to be reused by the next color-step to reduce data transfer between the device and the host through the PCI-E express bus.

### 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 PCI-E data transfers.

Figure 7 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 $V_4$ (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 $V_5$ of color-step 3 are assigned into the same color-chunk, which is a hybrid partition because $V_2$ and $V_5$ 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 $P$-steps, which means vertices and edges of color-step can be updated in parallel without concerning the data conflicts; for the $n$th 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.

### 4.3 Programming Interface

Frog offers a small set of user-defined APIs based on Medusa, and embraces a runtime system to automatically execute those APIs in parallel on the GPU. First, we provide device code APIs for developers to write GPU graph processing algorithms and present the data structure conveniently. Using these APIs (Structure APIs), programmers can define their computation on vertices, edges and messages. The vertex and edge APIs can also send messages.
to neighboring vertices. Second, Frog hides the GPU-specific programming details with a small set of system provided APIs (System APIs). Developers only need to define these functions, and the Frog runtime is responsible for executing these defined APIs in parallel on the GPU. Figure 8 shows an example of the PageRank implementation with Frog, which provides us a complete defined Structure APIs and System APIs. In the main function, Frog::Run(PageRank) invokes the PageRank function. We can update vertices of P-Steps in parallel without considering about any data race. As there are no adjacent vertices inside the color-chunk data of P-Steps. For kernel execution of S-step (function kernel_hybrid() as shown) to process the hybrid partition, we use atomic operations to make sure that no other threads can read or modify the value of the updating vertex. Then we send the latest value to its neighbor by the message buffer.

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 present the execution performance under our asynchronous approach execution, and 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.

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.

Figure 9 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.

5.1 Datasets and Testing Algorithms

- **Datasets**: We evaluate five real-world graphs with different properties in our experiments, as shown in Table 1. They are all real-world graphs from SNAP Datasets[14], which is supported by the project namely SNAP [1].

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Amazon</th>
<th>DBLP</th>
<th>WikiTalk</th>
<th>LiveJournal</th>
<th>Twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>735,322</td>
<td>986,286</td>
<td>2,394,385</td>
<td>4,847,571</td>
<td>41,652,229</td>
</tr>
<tr>
<td>Edges</td>
<td>5,138,012</td>
<td>6,707,236</td>
<td>50,021,410</td>
<td>98,475,391</td>
<td>1,486,365,167</td>
</tr>
</tbody>
</table>

**Frog** 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 [36]. This implementation of BFS explores all neighboring vertices from a starting vertex, in a level-by-level manner.

**Single Source Shortest Path (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.

**Connected Component (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.

**5.1.2 Experimental Environment**

We conduct our experiments on a Kepler-based GPU, NVIDIA Tesla K20m with 6GB main memory and 2688 CUDA cores. We compile the CUDA programs with CUDA 5.0 using the -arch=sm_35 flags. We run GraphChi [18], GraphLab [10] and Galois [17] on the system with 32GB main memory and 2 Intel Xeon E5620 CPUs at 2.40GHz. We directly reuse the source code of those engines from the authors.
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 Figure 9 (c), 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.

Figure 10 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 5 partitions as the baseline. This means that different numbers partitioning the generation graphs only have a little impact on our system, with the great performance improvement than other systems we get for comparison.

For two real-world graphs, WikiTalk and LiveJournal, both of them need a lot of colors to complete the coloring work, 56 colors for graph WikiTalk and more than 300 colors for graph LiveJournal. When we add a partition from \( k \) to \( k + 1 \), only a few vertices and edges are divided to them, most of vertices also stay in the last partition. This makes that there are only a small part of vertices in partition \( k \). The color-step invoked by these vertices obviously can not utilize the GPU resource. For example, when we split the graph LiveJournal into six partitions, the additional partition only holds 10.87% of vertices from the previous hybrid partition using five partitions. The total processing time of this additional partition, however, is nearly 30% of the previous time, which leads to a lower processing capacity (only 10.87% / 30% = 36% 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.

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. For the hybrid partition, there are a small number of vertices which should be processed sequentially or using atomic operations. We evaluate the overhead of last partition with 3-16 colors to partition four graphs. We also learn that the hybrid partition will not take more than 20% while using 5 or more colors for all the graphs under test.

In comparison to the only five colors used in the above cases, some large-scale graphs like twitter-2010 (the number of vertices is over 41 million) require to use more colors, because if we still use five colors to partition twitter-2010, the size of the largest color-chunk will be out of the device memory (the global memory size of device we used is 6GB). In our experiments, we adopt about 20 colors/partitions to process the large scale twitter-2010 graph, which cannot be processed by Duane Merrill’s approach [26], Medusa [36] and CuSha [16].

5.3 Performance of Asynchronous Approach

Asynchronous computations can substantially accelerate the convergence of many algorithms comparison with the BSP model. We evaluate four GPU-based systems as a comparison to our system. The first one is Medusa [36] which embraces an optimized runtime system with the simplified graph processing on GPUs based on the BSP model, but can only process graphs with scale smaller than device memory. The second one is Totem [7], a hybrid system that partitions the graph and processes tasks using both the traditional CPU cores and the GPU. There are two different graph partitioning strategies presented in Totem, HIGH-degree and LOW-degree. HIGH-degree divides the graph such that the highest degree vertices are assigned to the CPU, while LOW-degree divides the graph such that the lowest degree vertices are assigned to the GPU. Totem also can set the percentage of edges that assigned to different devices. In our experiments, we load all graph edges to the GPU device to evaluate the performance. The third one is CuSha [16], a vertex-centric graph processing framework focusing on exploring new graph representations named G-Shards and Concatenated Windows. We use the CW methods which gets the best performance of CuSha in our experiments. We also evaluate the performance of Merrill et al.’s work [26], which is the fourth one and actually only for BFS algorithm. We process all of the graphs using five partitions (except for the twitter-2010 graph) in both BFS and PageRank algorithms.

Table 2 presents the performance comparison between our asynchronous approach and four other systems running BFS algorithm. Our algorithm has clear performance gains as shown in the table. In particular, when processing the LiveJournal graph, our algorithm outperforms Merrill et al.’s method by 4.6X. From the Table 2, we also observe a performance degradation when processing Amazon graph by
we get a better performance than CuSha except for Medusa cannot process the graph LiveJournal at all. Our asynchronous approach can accelerate the convergence of PageRank with less iterations. Figure 11 shows that we can reduce more than 70% of iterations comparing to Medusa. Frogs gets a great improvement for random access on vertices. Frogs gets a great improvement for random access with the PSW method.

TABLE 2: Execution Time of BFS (in milliseconds)

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Amazon</th>
<th>DBLP</th>
<th>WikiTalk</th>
<th>LiveJournal</th>
<th>Twitter</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGC+1</td>
<td>3.320</td>
<td>2.906</td>
<td>3.995</td>
<td>32.436</td>
<td>null</td>
<td>–</td>
</tr>
<tr>
<td>Medusa</td>
<td>9.928</td>
<td>6.511</td>
<td>6.600</td>
<td>74.594</td>
<td>null</td>
<td>1.9X – 10X</td>
</tr>
<tr>
<td>CuSha</td>
<td>22.394</td>
<td>23.975</td>
<td>22.394</td>
<td>204.594</td>
<td>null</td>
<td>1.9X – 29X</td>
</tr>
<tr>
<td>GraphChi</td>
<td>593.6</td>
<td>733.3</td>
<td>925.6</td>
<td>2058.6</td>
<td>322.197</td>
<td>1.8X–291.7X</td>
</tr>
<tr>
<td>GraphLab</td>
<td>206.700</td>
<td>302.700</td>
<td>301.629</td>
<td>742.197</td>
<td>null</td>
<td>291.7X–291.7X</td>
</tr>
<tr>
<td>Galois</td>
<td>701</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:SGGT is the result of [26]. Null means that the system cannot process such dataset on GPU.

TABLE 3: Execution Time of PageRank (in seconds)

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Amazon</th>
<th>DBLP</th>
<th>WikiTalk</th>
<th>LiveJournal</th>
<th>Twitter</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frog</td>
<td>0.067</td>
<td>0.089</td>
<td>0.192</td>
<td>1.317</td>
<td>41.034</td>
<td>1.08X – 9.8X</td>
</tr>
<tr>
<td>SGC+1</td>
<td>0.065</td>
<td>0.087</td>
<td>0.185</td>
<td>1.307</td>
<td>null</td>
<td>1.08X – 9.8X</td>
</tr>
<tr>
<td>Medusa</td>
<td>0.282</td>
<td>0.574</td>
<td>0.609</td>
<td>null</td>
<td>null</td>
<td>4.2X – 29.7X</td>
</tr>
<tr>
<td>CuSha</td>
<td>0.062</td>
<td>0.513</td>
<td>0.990</td>
<td>1.870</td>
<td>null</td>
<td>0.92X – 5.8X</td>
</tr>
<tr>
<td>GraphChi</td>
<td>0.210</td>
<td>0.822</td>
<td>0.713</td>
<td>11.601</td>
<td>196.5</td>
<td>4.0X–5.8X</td>
</tr>
<tr>
<td>Galois</td>
<td>0.380</td>
<td>0.506</td>
<td>0.724</td>
<td>6.383</td>
<td>180.6</td>
<td>3.9X–5.8X</td>
</tr>
</tbody>
</table>

Note:We run 10 iterations for PageRank algorithm. Null means that the system can’t process such dataset.

We also get a good performance gain on PageRank, with an improvement of 4.2X better than Medusa. In particular, our asynchronous approach scales up to 29.7X on algorithm PageRank when processing the graph WikiTalk. We run the PageRank algorithm 10 iterations all the time. Table 3 shows us the performance. From the better performance we get based on both BFS and PageRank algorithms, our asynchronous approach can get a good performance against synchronized graph processing model.

We also run some experiments on PageRank by using a global termination condition, which confirms our asynchronous approach can accelerate the convergence of PageRank with less iterations. Figure 11 shows that we can reduce the number of iterations by over 70% than Medusa and CuSha, indicating much less overhead in the whole calculation process under our solution Frogs. By comparison, Medusa cannot process the graph LiveJournal at all.

For the synchronous graph processing framework CuSha [16], we get a better performance than CuSha except for one dataset while running PageRank. As an asynchronous execution system, Frogs exhibits a performance improvement from 3.7X to 29.1X over CuSha while running the BFS algorithm. Though CuSha partitions the graphs into different G-shards, which is similar with Frogs, it generates too many partitions with a lot of duplicate values to get a better performance.

We also evaluate three CPU based graph processing frameworks, GraphChi, GraphLab and Galois. GraphChi breaks large graphs into small parts, and uses a Parallel Sliding Windows (PSW) method to improve inefficiency of random access on vertices. Frogs gets a great improvement on BFS algorithm but only more than 4.0X improvement over GraphChi on PageRank algorithm. The reason is that PageRank needs so much random access and GraphChi improves a lot on random access with the PSW method. GraphLab [21] presents a framework based on the asynchronous computational model to achieve better conver-
Much research has been conducted on the efficient processing of large graphs due to the increasing number of graph-based applications. Some researchers used GPUs to accelerate graph algorithms [26], [11], because GPUs have significantly higher memory access bandwidth and much higher parallelism. Hong et al. [13] 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 [36] is a programming framework which provides a set of APIs for developers to implement their applications simply on GPUs. It 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. Totem [6] 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 do not fit the GPU memory size. CuSha [16] is a framework which makes it easy for users to define vertex-centric algorithms for processing large graphs on GPU.

### 6.2 Graph Partitioning

While processing large-scale graphs, partitioning is a major step which affects the performance of graph processing. A lot of work has been done on graph partitioning. PowerGraph [9] 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 communications. Tian et al. [32] introduce a graph-centric programming model that opens up the partition structure to users and allows information to flow freely inside a graph partition.

In order to process large graph data which cannot be loaded into GPU memory at a time, we also need to perform a partitioning strategy so that we can process subgraphs which fit the GPU memory. In addition, the rule for the partitioning method should also minimize the communication between different groups. Totem[6] provides two kinds of partitioning strategies to balance the offloading between CPU and GPU in order to make full use of them. And in the work [20], it uses a linear programming-based method to solve the incremental graph partitioning problem. Our partitioning strategy is based on our coloring method. We partition the graph into different subgroups by the color of vertices, and we can further partition the subgroup into smaller ones in order to fit into GPU memory. Both Hong et al. [13] and Medusa [36] 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. In order to solve this problem, Totem [6] 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. However, this method does not adapt to the asynchronous execution because of the data dependent computation on graphs.

### 6.3 Synchronous and Asynchronous Execution Models

Many of the advances in recent graph processing have focused on modeling the dependencies between data. As a consequence, many graph parallel abstractions, like Pregel and GraphLab, naturally express computational dependencies. Many important graph algorithms iteratively update a large set of vertices or edges; these updates depend on values of other adjacent vertices or edges. 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 [22], it demonstrates that how asynchronous computation can substantially accelerate the convergence of some algorithms like PageRank. Hong et al. [13] observe a nature of the parallel BFS algorithm that the amount of available parallelism is associated with the execution level. Based on this nature, this work presents a BFS implementation on a GPU-accelerated platform and divides the processing into two phases. The phase on the GPU starts once enough parallelism is exposed.

Pregel [23] is a vertex-centric framework based on the Bulk Synchronous Parallel (BSP) model, which uses synchronous super-steps for computation and communication. Mizan [15] uses fine-grained vertex migration to achieve the workload balance across Pregel super-steps. Similar to systems such as Pregel, X-Stream [29] 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 [17] 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 [31] 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 [2].

GraphLab [21] presents a framework based on the asynchronous computational model to achieve better convergence than the synchronous implementation. Distributed GraphLab [22] extends the GraphLab framework to the substantially more challenging distributed setting while preserving strong data consistency guarantees and provides two substantially different approaches to ensure data consistency, chromatic engine and locking engine. The PowerGraph framework [9] combines the shared-memory and asynchronous properties of GraphLab with the associative combining concept of Pregel. To exploit asynchronous parallelism in iterative algorithms on distributed shared memory, ASPIRE [33] 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 Bulk Synchronous Parallel (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% of the entire graph) into a moderate number of partitions without no 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. Our experiments show that our system Frog provides about 3.7X improvement over CuSha [16] and 1.9X–10.6X improvement over Medusa [36] on BFS algorithm.
- 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 GraphChi [18], while the other systems like [26], Medusa [36] and CuSha [16] are unable to run such scale graphs.

As for the future work, we plan to support dynamic graph processing. In addition, we also plan to implement a hybrid graph processing platform to efficiently use both the traditional CPU cores and GPU(s) like Totem [6].

### ACKNOWLEDGMENTS

We want to thank Xiaosong Ma of the NCSU, Zuoning Yin of the GraphSQL, Bing Bing Zhou of the USYD, and Pingpeng Yuan of the HUST for their valuable comments.

### REFERENCES


