Feluca: A Two-Stage Graph Coloring Algorithm with Color-centric Paradigm on GPU

Zhigao Zheng, Xuanhua Shi†, Ligang He, Hai Jin, Shuo Wei, Hulin Dai, Xuan Peng

Abstract—There are great challenges in performing graph coloring on GPU in general. First, the long-tail problem exists in the recursion algorithm because the conflict (i.e., different threads assign the adjacent nodes to the same color) becomes more likely to occur as the number of iterations increases. Second, it is hard to parallelize the sequential spread algorithm because the color allocation depends on the adjoining iteration. Third, the atomic operation is widely used on GPU to maintain the color list, which can greatly reduce the efficiency of GPU threads.

In this paper, we propose a two-stage high-performance graph coloring algorithm, called **Feluca**, aiming to address the above challenges. Feluca combines the recursion-based method with the sequential spread-based method. In the first stage, Feluca uses a recursive routine to color a majority of vertices in the graph. Then, it switches to the sequential spread method to color the remaining vertices in order to avoid the conflicts of the recursive algorithm. Moreover, the following techniques are proposed to further improve the graph coloring performance. i) A new method is proposed to eliminate the cycles in the graph; ii) a top-down scheme is developed to avoid the atomic operation originally required for color selection; and iii) a novel color-centric coloring paradigm is designed to improve the degree of parallelism for the sequential spread part. All these newly developed techniques, together with further GPU-specific optimizations such as coalesced memory access, comprise an efficient parallel graph coloring solution in Feluca. We have conducted extensive experiments on NVIDIA GPU. The results show that Feluca can achieve 1.19 – 8.39× speedup over the state-of-the-art algorithms.

Index Terms—Graph Coloring, GPGPU, Parallelism, Color-centric Paradigm, Pipeline.

1 Introduction

Given a graph $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges. Two nodes $v_1, v_2 \in V$ are regarded as being adjacent to each other if $(v_1, v_2) \in E$, i.e., an edge exists between v_1 and v_2. Let C be a set of colors. Graph coloring is a task of assigning each vertex $v \in V$ a color $c \in C$ such that there are no two adjacent vertices which have the same color and the number of different colors used, $|C|$, is as small as possible. Graph coloring is widely applied to the problems such as resource allocation and scheduling, time-tabling [1], register allocation & spilling [2], and puzzle-solving (Sudoku) [3].

The smallest number of colors that is needed to color a graph is called *Chromatic number,* $\chi(G)$ [4]. Determining $\chi(G)$ is an "NP-complete" problem. Hence, a practical approach to such a computationally intractable problem is to relax the optimality constraint and find the "near-optimal" solution [5]. Suppose A is a coloring algorithm, and $A(G)$ is the number of colors used by algorithm A on graph G. The near-optimal solution can be defined as: finding an efficient coloring algorithm A such that $A(G)/\chi(G)$ is close to 1. Several research studies have been conducted to use the heuristics and greedy approaches to perform near optimal coloring [6]. However, due to the burgeoning size of real-world graphs, even the algorithms with the linear times need to resort to parallel computing to achieve practical solving times. Furthermore, the solving speed (i.e., performance) of a near-optimal algorithm typically contradicts its solving quality. Therefore, it is important to strike a balance between performance and quality for the designed algorithms. This paper aims to design an efficient parallel coloring algorithm that is able to find near optimal solutions.

The Graphic Processing Unit (GPU) is a promising device to accelerate graph coloring on large scale graphs, thanks to its massive degree of parallelism and high memory access bandwidth. However, inherent issues in graph processing such as random memory accesses and workload imbalance make it very challenging to fully utilize the parallel computing power of GPU [7], [8], [9], [10], [11], [12], [13]. A significant amount of work has been carried out to develop new data layout models [14], graph programming models (GAS, BSP), memory access patterns, workload mapping in order to optimize graph processing on GPU [15], [16], [17], [18], [19]. In graph coloring, although recent attempts [20], [21] have been made, unleashing the full power of GPU to achieve high-performance graph coloring still remains a great challenge.

In this paper, we propose a two-stage graph coloring algorithm, called **Feluca**, which is custom-designed to unleash the full potential of GPU. In the first stage, Feluca adopts a recursive coloring algorithm to color a majority of vertices

- *Zhigao Zheng, Xuanhua Shi, Hai Jin, Shuo Wei, Hulin Dai and Xuan Peng are with National Engineering Research Center for Big Data Technology and System / Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China.*
  E-mail: {zhengzhigao,xhshi,hjin,weishuo,hulindai,piecesix}@hust.edu.cn
- *Ligang He is with the Department of Computer Science, University of Warwick, United Kingdom.*
  E-mail: ligang.he@warwick.ac.uk

†: *Corresponding Author.*
Manuscript received Month Date, Year; revised Month Date, Year.

in a small number of iterations. To avoid the long tail phenomenon with the recursive coloring approach, Feluca switches to the sequential spread approach in the second stage. Moreover, Feluca proposed the following techniques to further improve the graph coloring performance and comprise an efficient parallel graph coloring solution.

1) We present a ***two-stage coloring algorithm on GPU, Feluca,*** which combines the recursive approach (the first stage) with the sequential spread approach (the second stage).
2) We design a ***cycle elimination method*** to ease the process of spreading the color value in Feluca. Specifically Feluca changes the directed edge $< v_i, v_j >$ to $< v_j, v_i >$ if $i > j$, so as to eliminate the cyclic paths in a graph and avoid the infinite loops caused by cyclic sub-graphs. Based on the cycle elimination technique, we design a ***top-down color selection scheme*** to select the suitable colors from the color array sequentially. Most existing color selection schemes generate many *atomic* operations to ensure the correctness of the algorithms. In order to improve the efficiency of color selection, we propose a continuous top-down color selection scheme to select next color of the current vertex for the conflicting vertex.
3) We design a ***color-centric paradigm*** to improve the degree of parallelism for the sequential spread stage. We allocate thread block(s) to process a color and organize these blocks in pipeline. With this pipeline mechanism, the results of the $(i-1)^{th}$ iteration can be easily used by the i^{th} iteration.
4) We design a set of evaluation schemes for Feluca. The experimental results show that Feluca outperforms the existing state-of-the-art algorithms by up to 8.39× on GPU.

The rest of this paper is organized as follows: Section 2 demonstrates the performance problem of graph coloring algorithm on GPU and discusses the motivation of this research. Then the algorithm design is presented in Section 3. Section 4 presents the optimization strategies for the proposed graph coloring algorithm. The overall performance of Feluca is evaluated in Section 5. Section 6 discusses the related work and Section 7 concludes the paper and discusses future research opportunities.

2 MOTIVATION

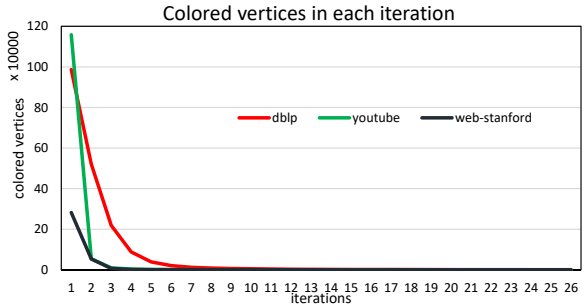
In this section, we demonstrate the main problem when running the graph coloring algorithms on GPU, which motivate us to develop Feluca.

According to previous studies, the graph coloring algorithms can be divided into several categories, in which *sequential spread* and *recursion* are most often used [22], [23], [24].

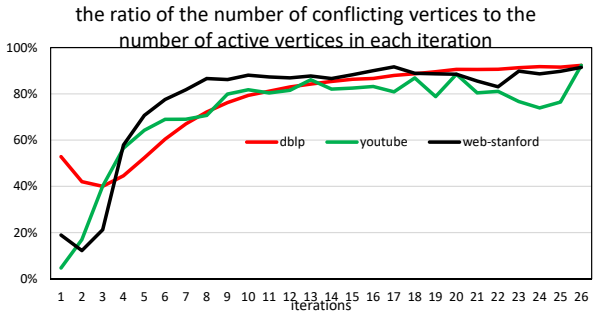
The basic idea of the sequential spread algorithm is to traverse the entire graph using the algorithms such as greedy coloring, and check the vertex's color one by one. These algorithms proceed in synchronized steps and use the threads to work on the active vertices. A crucial attribute of a synchronous computing model is the number of synchronized steps. An algorithm that has a less number of synchronized steps delivers the better performance.

In the sequential spread model, each synchronized step can be divided into three phase. In the first phase, the colored vertices need to send their colors to their neighbouring vertices. In the second phase, the neighbours receive the message about the colors and then in the third phase, the neighbours execute the local computation. However, we observed from our benchmarking experiments that although the number of synchronized steps is large, there are only a small number of active vertices in each step. Moreover, it is difficult to run this execution model in parallel, because the active vertices in current iteration can be colored only when the results of previous iterations are known.

The other type of graph coloring algorithms employs the recursive execution, which works in a similar way as PageRank. High performance recursion-based graph coloring algorithms have been developed in recent works [20], [25]. In this coloring model, every vertex is assigned a color at the beginning. Then every vertex sends its color value to its neighbours. Next, the neighbouring vertex updates its color if the color conflicting occurs (i.e., the neighbouring vertices have the same color).



(a) Colored vertices



(b) Vertices conflict rate

Fig. 1: The number of colored vertices (Figure 1a) and the ratio of the number of conflicting vertices to the numberof active vertices (figure 1b) as the graph coloring process progress through each iteration.

The colors of most vertices converge in the first few iterations. The remaining small fraction of vertices take a long time to converge to the final color due to color conflicts. This *long-tail* phenomenon can be demonstrated from the following benchmark experiments. We ran the experiments on a NVIDIA Tesla P100 GPU, which is equipped with

16 GB on-board memory, 3,584 CUDA cores, and Red Hat Enterprise Linux Server release 6.2 (Linux version 3.10.0-514.el7.x86_64). As shown in figure 1a, the number of vertices that are colored in each iteration drops dramatically in the first few iterations (i.e., these vertices converge to their final colors) and only a very small fraction of vertices are active in a large number of remaining iterations. Take the youtube dataset as an example. There are 1,157,828 vertices in total. 1,148,571 vertices converge in the first 2 iterations, while the remaining 9,257 vertices take more than 30 iterations to converge. We also plot the ratio of the number of conflicting vertices to the number of active vertices in each iteration, shown in figure 1b. The ratio is only 4% in the first iteration, and increases to 92.3% in the 26^{th} iteration. These results suggest the recursion algorithm is very effective in the first few iterations, but does not conduct much useful work in the remaining iterations.

We have shown that a single coloring strategy in the entire execution may be problematic because the conflict always exists and can be dramatically different in each iteration. This motivates us to develop an adaptive and hybrid algorithm for graph coloring, where the coloring mode changes adaptively and automatically at different stages, aiming to achieve the best overall performance.

3 Algorithm Design

This section presents the design philosophy and describes the algorithm in Feluca.

3.1 Design Philosophy

Graph coloring is widely used to partition the connected tasks in many parallel applications. The connected tasks are partitioned into independent tasks, which can then be executed concurrently. There are two basic objectives in the graph coloring work: a) reducing the number of colors, and b) improving the partition speed. If the tasks represented by the vertices in the graph are computationally expensive, then typically the objective of the graph coloring is to use as few colors as possible. Since the tasks with the same color can be run in parallel, the graph coloring with this objective can help achieve the highest average degree of parallellism for the set of tasks in the graph. This objective represents the effectiveness of the coloring algorithm. A large amount of works have been carried out regarding this coloring objective [26], [27], [28].

On the other hand, if the tasks in the graph are fairly small and one has to find new graph coloring results repeatedly, then the time spent by graph colorings may take up a significant portion of the entire computation time. In these types of application, the coloring efficiency is more important than the coloring effectiveness. There are some works with respect to this objective recently [29], [30], [31], [32]. In this paper, we focus on the second objective.

In this work, we present a high performance graph coloring algorithm on GPU, the fundamental idea of which is to combine the advantages of the recursion and the sequential spread model and avoid their drawbacks (as discussed in the motivation section). Hence, the target of our algorithm is to maintain the coloring effectiveness (resulting in a coloring plan no worse than the existing research) while improving the coloring efficiency (coloring the graphs faster than current methods). The fundamental strategy of our method is to color as many vertices as possible in a round and avoid the conflicts.

In order to meet the above objectives, There are two stages in Feluca. It starts with the recursion execution model, which can color a majority of vertices in the first few iterations, and then switches to the sequential spread execution model once there are too many conflicts occurring. We also propose a novel color-centric coloring paradigm to improve the degree of parallelism in the sequential spread stage.

Further, we develop several optimization techniques including the top-down coloring scheme and the cycle elimination method, which is presented in Section 4.

3.2 Two-Stage Graph Coloring Algorithm

The coloring processing starts on the host. The color array and the graph topology data are then loaded on GPU for coloring. The two-stage graph coloring algorithm is outlined in algorithm 1.

Algorithm 1 Feluca: A High-Performance Graph Coloring Algorithm

Require: Graph, G; $fraction$
Ensure: Graph coloring plan, $COLORS$; and the colors $color\_num$ used for coloring graph G;

```
1: function RecursionExec(G)
2:     C(G) ← init_color_randomly(G);
3:     while coloredvertices/Totalvertices ≤ fraction do in parallel
4:         while v_j ∈ V_i && i < j do
5:             if c_i == c_j then
6:                 update c_j;
7:             end if
8:         end while
9:         if coloredvertices/Totalvertices > fraction then
10:            while v_j ∈ V_i && i < j do
11:                if c_i == c_j then
12:                    ConflictQueue.enqueue(v_j);
13:                end if
14:            end while
15:            SequentialExec(ConflictQueue);
16:        end if
17:    end while
18: end function
19: function SequentialExec(ConflictQueue)
20:    while ConflictQueue do in parallel
21:        while c ∈ C and c is available do
22:            if v_i ∉ nbor(v_j) then
23:                ConflictQueue.dequeue(v_i);
24:            end if
25:        end while
26:    end while
27: end function
```

In Feluca, we maintain a read-only color array, denoted by *COLORS*, which can be visited by all threads. The parameter $fraction$ is the colored vertex rate in the recursion stage, which can be set by the user or the algorithm itself. In

Section 5, the experiments are carried out to obtain a suitable value of $fraction$. In our algorithm, we initialize the colors of all the vertices in *COLORS* randomly. We define V_i as the set of neighbours of vertex v_i and c_i as the color of v_i. In the recursion loop, vertex v_i broadcast its own color c_i to its neighbours in V_i following the edges' directions. Once vertex $v_j \in V_i$ receives the color from vertex v_i, it compares its color c_j with c_i. The comparisons conducted by different vertices are conducted in parallel. If $c_j = c_i$, v_j selects a new color from the *COLORS* array and updates c_j. The process repeats until the colors of all vertices in V_i are different with the color of v_i.

In the sequential spread stage, Feluca generates a block of threads and scans the remaining vertices in parallel to find the suitable vertices for each color. In order to improve the degree of parallelism for this algorithm and avoid the conflicts, Feluca assigns a block of threads for each color. The thread blocks for different colors are put into execution in a pipeline. The late blocks can use the coloring results of the early blocks in the pipeline. By doing so, the conflicts in the sequential spread model can also be avoided.

N_i and t_i denote the colored vertices and the time spent in iteration i. $s_i = N_i/t_i$ can then be used to express the coloring speed in iteration i. The coloring rate (i.e., the percentage of the vertices that have been colored) up to iteration i can be expressed by Equation 1, where N is the total number of vertices in the graph.

Feluca switches to the sequential spread coloring method once the color rate (λ) in the recursion stage is lower than the value of the parameter $fraction$. T denotes the total coloring time.We can have $T = \sum_{i=1}^{r} t_i + \sum_{j=r+1}^{r+q} t_j$, where r is the number of iterations in the recursion stage while q is the number of iterations in the sequential spread stage. Hence, we can express the coloring time as formula 2.

$$\lambda = \frac{\sum_{j=1}^{i} N_j}{N} \tag{1}$$

$$T = \sum_{i=1}^{r} t_i + \sum_{j=r+1}^{r+q} t_j = \sum_{i=1}^{r} \frac{N_i}{s_i} + \sum_{j=r+1}^{r+q} \frac{N_j}{s_j} \tag{2}$$

$$\sum_{i=1}^{r} \frac{N_i}{s_{rmax}} + \sum_{j=r+1}^{r+q} \frac{N_j}{s_{smax}} \leq T \leq \sum_{i=1}^{r} \frac{N_i}{s_{rmin}} + \sum_{j=r+1}^{r+q} \frac{N_j}{s_{smin}} \tag{3}$$

Furthermore, we use s_{rmin}, s_{rmax} and s_{smin}, s_{smax} to denote the minimum and maximum value of the coloring speed at recursion and sequential spread stage, respectively. Then we can express T as formula 3.

Combining equation 1 and formula 3, we can have formula 4.

$$\frac{\lambda \times N}{s_{rmax}} + \frac{(1-\lambda) \times N}{s_{smax}} \leq T \leq \frac{\lambda \times N}{s_{rmin}} + \frac{(1-\lambda) \times N}{s_{smin}} \tag{4}$$

Formula 4 indicates that T can be formulated as the form of function over λ shown in formula 5, where s_r and s_s are the coloring speed in the recursion stage and sequential spread stage, and t_r t_s are the coloring time spent in the recursion stage and sequential spread stage, respectively, and $f_1(t_r)$ and $f_2(s_s)$ are the functions that takes s_r (or t_r) and t_s as input, respectively.

$$T = \lambda f_1(s_r) + (1-\lambda) f_2(s_s) = \lambda f_1(\frac{N}{t_r}) + (1-\lambda) f_2(\frac{N}{t_s}) \tag{5}$$

It can be seen from equation 5 that finding a minimum T is a convex optimization problem. When the color rate λ is higher than $fraction$, the graph coloring in Feluca switches from the first stage (recursion) to the second stage (sequential spread). In section 5, we will carry out the experiments to obtain a suitable threshold value $fraction$ for λ.

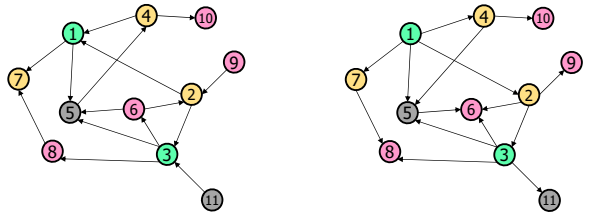
4 Optimization Techniques

In this section, we present the optimization techniques to improve the algorithm efficiency.

4.1 Cycle Elimination Method and Top-down Coloring Scheme

When the recursion-based coloring algorithm works on cyclic graphs, it is easy to fall into an infinite loop since the algorithm runs following the edge's direction. In order to solve this problem, we develop a method in Feluca to eliminate the cyclic paths in the directed graphs. In the method, Feluca checks all the vertices $v_j \in V_i$ (i.e., the set of neighbours of vertex v_i) before coloring v_i, and changes the edge $< v_i, v_j >$ to $< v_j, v_i >$, if $i > j$. For example, the edge $< v_4, v_1 >$ in figure 2a is changed to $< v_1, v_4 >$ in figure 2b. The change of edge direction does not affect the coloring plan. For example, these two coloring scheme in figure 2a and figure 2b are regarded as same.

The color selecting scheme is the most important part of a graph coloring algorithm. A sophisticated color selecting scheme can reduce the number of colors. However, it is often hard to parallelize these color selecting algorithms, and the color conflicting may slow down the coloring process. On the contrary, a simple color selecting algorithm may be parallelized easily, but often leads to a huge color set in the final coloring plan.



(a) Cyclical graph coloring (b) Acyclic graph coloring

Fig. 2: Coloring scheme for cyclical graph and acyclic graph.

In Feluca, a top-down color selection scheme is proposed to select the suitable color for the conflicting vertices. This scheme can avoid the atomic operations fully. As presented earlier in this section, we have transformed the graph to a directed acyclic graph (DAG). Our top-down coloring scheme starts coloring from the top level (root) of the graph, and traverses the graph in the same way as the BFS (Breadth-

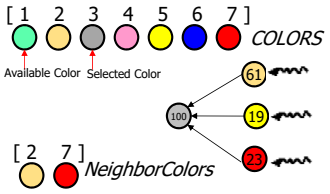


Fig. 3: The top-down coloring scheme in Feluca. "*COLORS*" and "*NeighborColors*" are the color array for the whole graph and the temporary array for the colors which are just used in iteration $i-1$, respectively. The "Selected color" is the color selected by Feluca for vertex 100 in iteration i. The "Available Color" is the first color which can be used to color vertex 100.

First-Search) algorithm. There are often multiple vertices in a graph level, which are colored by multiple threads in parallel. After a level of vertices are colored, the coloring scheme moves onto the next level in the DAG graph (hence a top-down coloring scheme), and the computation moves into next iteration. The process repeats until all vertices have been colored.

A color array *COLORS* is used to hold all used colors, and a temporary array *NeighborColors* with the fixed length is used to hold the colors which are used by the neighbours of the current vertex. In our implementation, we allocate two bytes for the NeighborColors array to store the minimum and maximum color (i.e., the colors with minimum and maximum IDs) of the neighbors. By only traversing the *COLORS* array once Feluca can find the candidate color for current vertex in the COLORS array from the minimum color to the maximum color recorded in the *NeighborColors* array. When a new color has to be used for a vertex, the new color is appended to the end of the color array. After a thread has colored a vertex in a level, Feluca finds the neighbours of a vertex following its outgoing edge. The neighbours are the vertices in the next level, which are also the vertices ready to be colored in next iteration. When a thread is trying to color a vertex in an iteration (suppose in iteration i), it checks the array *NeighborColors*, which is constructed and assigned to the vertex's parents in last iteration (iteration $i-1$), in the following way. Suppose the length of *NeighborColors* is len, Feluca goes through *NeighborColors*. If there exists a color $c = NeighborColors_j + 1, 0 < j < len$, where $c < NeighborColors_{len-1}$, then Feluca will assign color c to this vertex. Otherwise, Feluca assigns to this vertex the color that is immediately after the last color among all parents' colors in the color array.

An example is illustrated in figure 3 to show how our top-down coloring scheme works. In figure 3, suppose the graph coloring is currently in iteration i, the array *NeighborColors* with the fixed length of 2. It can be seen from figure 3 that vertices 19, 23 and 61 are the parents of current vertex 100. Therefore when the coloring scheme colored vertices 19, 23 and 61 in iteration $i-1$ (assume the colors assigned to vertices 19, 23 and 61 are the 2nd, 5th and 7th color in the color array *COLORS*, respectively, as shown in figure 3), it followed the edges $< 19, 100 >$, $< 23, 100 >$ and $< 61, 100 >$ to find that vertex 100 is a vertex to be colored in iteration i. Now suppose the coloring scheme is trying to color vertex 5 in iteration i. Our coloring scheme realizes that vertices 19, 23 and 61 are the parents of 100. Then, it stores the colors that are just used into *NeighborColors* (*NeighborColors* stores all colors that are just used if it is long enough; otherwise, it stores the colors that has the lowest and highest color number). Our coloring scheme will check if the color $NeighborColors[j]+1$ is in *NeighborColors* or not, where $0 \leq j \leq NeighborColors.length - 1$. It will assign the color $NeighborColors[j] + 1$, which is the 3rd color in the color array, to vertex 100 when $NeighborColors[j]+1 \notin NeighborColors$. Our coloring scheme only needs to check the colors of the parents assigned in last iteration, which is stored in *NeighborColors*, to find a suitable color for the current active vertex, while the traditional coloring scheme would search the whole color array to find the first available color, which would assign the 1st color in the color array *COLORS* (labeled with "Available Color" in figure 3) to vertex 100 in this example. Our color chosen scheme only focuses on the current vertex and its parent vertices colored in last iteration. This design enables the GPU threads to update the colors of their current vertices in parallel without atomic operation/lock.

4.2 Color-centric Coloring Paradigm

When GPU is used to accelerate graph processing, threads in GPU are organized in a grid and all the threads in a grid execute the same kernel function. The threads running a kernel are organized in a two-level hierarchy: a grid consists of a number of thread blocks and each block comprises a set of threads. It is a great challenge to parallelize the sequential spread coloring, because coloring the vertices in iteration i depends on the results of iteration $(i-1)$. In order to improve the degree of parallelism of the sequential spread stage in Feluca, we proposed a new coloring scheme, called the color-centric scheme. The traditional algorithm for sequential pread coloring is vertex-centric, i.e., finding a suitable color for each vertex. In our color-centric scheme, we find all suitable vertices for each color, which is presented in detail next.

After the first coloring stage (the recursion stage) is finished, we record all the remaining vertices which have not converged to the final colors yet (called uncolored vertices). In the color-centric scheme, for each color, we generate a thread block to find in the list of remaining un-colored vertices all vertices that can be assigned with this color. A thread block starts with the uncolored vertices in the first graph level, and moves down the graph levels until all uncolored vertices have been colored.

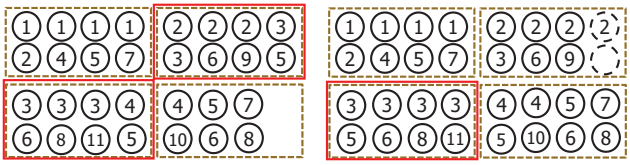
In the color-centric scheme, different thread blocks find vertices for different colors in parallel. We develop two parallelization strategies to run the thread blocks for each color. In the first parallelization strategy, we set the number of colors needed for the remaining uncolored vertices. We then generate a thread block to find all vertices for each color. In particular, a thread block for a color collectively find all vertices that do not have direct links between any two of them and assign all these vertices to this color. We start the execution of all thread blocks at the same time. In this strategy, it is possible that different thread blocks assign different colors to the same vertex. When this

happens, the vertex keeps the color which has the smallest color ID among the conflicting colors. The shortcoming of this parallelization strategy is that we have to first set the number of colors for the remaining uncolored vertices. We cannot accurately know the exact number of colors needed. On one hand, if the number of colors is set too low, it is impossible to deliver the final coloring plan which does not contain conflicting. On the other hand, if the number is set too high, the number of colors in the final coloring plan will be much higher than that in the optimal coloring plan.

In the second parallelization strategy, we do not start all thread blocks at the same time, but run the thread blocks in pipeline. In particular, we first start the thread block for the first color. The thread block starts with find all uncolored vertices in the first level of the graph that can be assigned to the first color. After the first level is processed, the thread block moves to the second level and repeats the process. While the thread block for the first color is processing the second level, we start the thread block for the second color and start to find all remaining uncolored vertices that can be assigned to the second color. Similarly, when the thread block for the first color moves to the third level, the thread block for the second color moves to the second level and the thread block for the third color start processing the first level. The pipeline goes on until all uncolored vertices after the recursion stage have been colored. In the second strategy, we do not have the problem that we may set the number of colors too high. When all vertices have been colored, the pipeline stops and the number of used colors then is the number of colors in the final coloring plan. Our experiments show that the first and second parallelization strategies have similar running performance. But the second strategy typically uses a fewer number of colors than the first strategy. Therefore, we use the second parallelization strategy in the sequential spread stage in Feluca.

4.3 The Edgelist Graph Representation

A GPU can reach its peak memory access bandwidth only when the algorithm has a regular memory access pattern, i.e., the data accessed by the consecutive threads of a warp occupies the contiguous memory locations. When there is not a regular memory access pattern, a naive solution to improve the GPU memory access efficiency is to sort the edges by source vertex ID, which is shown in figure 4a. Figure 4a shows the edgelist representation of the graph in figure 2. The graph has 11 vertices and 15 edges. The 15 edges are partitioned into 4 edge blocks. In this partition, the edges are sorted in the order of their source vertex IDs and every edge block contains 4 edges. Suppose that a warp contains 4 threads. Then a warp can process the whole block. But in this partition, the edges with source vertex ID 3 are allocated to 2 blocks. Hence, the threads processing the edges with source vertex 2 in block 2 need to wait for the threads which process the edges with source vertex ID 3. It is also the case for the threads which process with source vertex 4. When processing the real-world graphs, the edges of high degree vertices may be scattered in several blocks. On the other hand, a block may contain the edges from more than one low-degree vertex. Under this circumstance the computing load of different edge blocks vary, which will cause the threads in a thread block to wait for other thread blocks with more computing load.



(a) The edgelist representation of the graph in figure 2.

(b) The ordered graph with virtual edges.

Fig. 4: An example of ordering the graphs by adding virtual edges.

In order to address this problem, Feluca adds some virtual edges, which do not take any memory space, in the edgelist-based graph representation such that either the edges in an edge block have the same source ID or an edge block contains all edges from different vertices.

For example, We add a virtual edge with source vertex ID 2 as shown in figure 4b. In the ordered graph, all the edges with source vertex ID 1, 2 and 3 are located in a single warp and all the remaining edges are located in the same warp (i.e., the warp contains all edges for vertice 4, 5 and 7). This representation can avoid the overhead that the threads wait between the edges with source vertex 2 and 3. As edges are sorted in the order of the source vertex ID in the grid, continuous and coalesced memory access pattern can be achieved. By using this method the threads for running the virtual edges are figure 4a, two warps (marked by the red box) are needed when coloring vertex 3. But in these two warps, the four threads which work on the edges with source vertices 2 and 4 are not performing the effective work (since those edges have been processed in previous rounds). In figure 4, after we add the virtual edge, only one warp is needed to color vertex 3, and there is no idle thread. On the other hand, adding virtual edge can not increase overall run time since the bottleneck of the data transfer from host to GPU is the PCI-E. The graph is loaded to the GPU onboard memory through PCI-E bus but the PCI-E bandwidth is much lower than the memory access in CPU and GPU (the bandwidth for PCIe 3.0 is 32GB/s, while the memory bandwidth of the NVIDIA K20m GPU is 208 GB/s, the memory bandwidth of NVIDIA P100 GPU is 720GB/s).

GPUs have been successfully used for in-memory graph processing systems [33], [34], [35]. These systems can achieve up to two orders of magnitude of speedup over the state-of-the-art CPU-based graph systems [34], [36]. These execution models can only process the graphs that are smaller than the GPU memory (i.e., the entire graph can be loaded into the GPU memory). However, GPUs have limited memory space. NVIDIA Tesla K20m and P100 are the most typical GPU accelerator, which are widely used in HPC and some other application areas. The global memory of NVIDIA Tesla K20m is 5 GB, while P100 has 16 GB memory. However, many real-word graphs have millions of vertices and edges which are too large to fit into the GPU memory. Table 1 in section 5 lists eight graphs used in this paper, which also often used by the literature [34], [35], [37]. Suppose that storing an edge consumes 8 bytes, and a vertex

ID consumes 4 bytes. Then 12 GB and 16 GB memory are needed to hold the topology data of `random-graph` and `webbase-2001` graph, respectively. Since more memory space will be used to store the attribute data during the graph processing, the actual memory consumption is even larger. So, how to load a large scale graph into GPU is a great challenge.

To process the graphs bigger than the GPU memory, we partition the graphs into several blocks and then assign the blocks to the GPU in a pipeline fashion. In Feluca, we begin to color the graph block once it loaded into the GPU, and we also begin to transfer the next graph block to the GPU at the same time. By doing so, we overlap the computation with communication. This is a widely used approach in most large-scale graph processing systems on GPU, such as Frog [35] and Gunrock [36]. Benefit from the edgelist-based graph presentation, it is easy to partition the graphs into several transport streams. By using this model, Feluca can transfer graph data from host memory to GPU memory simultaneously with the computation, which enable Feluca to process large-scale graphs on a single GPU.

5 Evaluation

In this section, we present the results of experimental evaluation for different design choices in Feluca and compare Feluca with the state-of-the-art graph coloring techniques on both power-law and random graphs.

5.1 Experimental Setup

We have conducted the experiments with both directed and undirected graphs. (u, v) represents the undirected edge between vertices u and v while $< u, v >$ represents a directed edge from u to v.

TABLE 1: Datasets used in the experiments

Datasets	Vertices	Edges	Direction
Stanford	281,903	2,312,497	Directed
dblp	986,207	6,707,236	Undirected
youtube	1,157,828	2,987,624	Undirected
RoadNet	1,971,282	5,533,214	Undirected
Wiki	2,394,385	5,021,410	Undirected
soc-lj	4,847,571	68,993,773	Directed
RMAT	9,999,993	160,000,000	Undirected
random	19,999,888	100,000,000	Undirected
twitter	41,652,230	1,468,365,182	Directed
webbase	118,142,155	1,019,903,190	Directed

We have carried out the experiments on total of 10 different graphs, 8 real-world graphs and 2 synthetic graphs as detailed in table 1. The vertex degree among all graphs ranges from 2 to 10^6. Synthetic graphs `RMAT` and `random` are generated using PaRMAT [37] and have the random degree distribution. The `twitter` and `webbase` are shared in the Laboratory for Web Algorithmics (LAW) [38] and the remaining graphs are obtained from Stanford Network Analysis Project (SNAP) [39].

We conduct the experiments on a `NVIDIA Tesla P100 GPU`, which is a Pascal architecture-based GPU equipped with 16 GB on board memory and 3,584 CUDA cores. The GPU is coupled with host machine equipped with 2 Intel(R) Xeon(R) E5-2670 CPUs, each at 2.60 GHz, and 8 GB memory. The host machine is running RedHat OS version 4.4.5-6. The algorithm is implemented with C++ and CUDA 9.0 using the "-arch=sm35" compute compatibility flag.

5.2 The Algorithms for Comparison

We compared Feluca with three state-of-the-art methods in the experiments, which are Kokkos, JPL and cuSPARSE. All these three methods are implemented on NVIDIA GPU. We describe some operation details of these three methods as follows.

- **Kokkos**. Kokkos uses a first-fit policy to assign color for the vertices [20]. In the first-fit algorithm, a large *FORBID* array is used to store the colors of the neighbors of the current coloring vertex. Namely, the current coloring vertex cannot choose the colors from the *FORBID* array. This method can achieve good processing speed. But the *FORBID* arrays of the large degree vertices consume large memory space. On the other hand, if a small *FORBID* array is used for for the large degree vertices, multiple memory accesses are then needed, which slows down the coloring process. In Feluca, we use an array *COLORS* to hold all used colors, and a temporary array *NeighborColors* with a fixed length is used to hold the colors which are just used by the neighbors of the current vertex. In our method, we can find the candidate color for current vertex in the *COLORS* array from the minimum number of *NeighborColors* to the maximum number of *NeighborColors* by just access *COLORS* one time.
- **Jones-Plassmann-Luby (JPL) Graph Coloring Algorithm**. The JPL coloring algorithm uses an approximate maximal independent set policy to partition the vertices into several sets and assign the colors to each set [40]. This coloring policy uses the iteration approach, which can be trapped in the long-tail problem easily.
- **cuSPARSE**. The cuSPARSE library is developed by NVIDIA. The coloring implementation of cuSPARSE follows a csrcolor [40] routine by using a *multi-hash* method to find the independent sets. Similar as the JPL graph coloring algorithm, it is easy to be trapped in the long-tail problem. Feluca uses the iteration execution only to color part of the vertices quickly, and then switches to the sequential spread stage. The second stage adopts the color-centric method, which enables higher degree of parallelism and consequently further reduces the execution time (see Figure 6).

5.3 Recursion vs. Sequential Spread

In contrast to majority of existing solutions that adopt purely recursion based approach or sequential spread based approach, Feluca combines the both approach into a single solution. In this section, we empirically evaluate the strengths and weaknesses of both approach. As explained in Section 3, Feluca utilizes a parameter called $fraction$ to control the switching from recursion based processing to sequential spread based approach. The fraction is the

ratio of number of vertices already colored to total number of vertices in the graph. Setting fraction to 0.0 makes the system purely sequential while setting fraction to be 1.0 makes it fully recursive algorithm.

TABLE 2: Execution time for different coloring algorithms (in milliseconds)

Datasets	Recursion Only		Sequential Spread Only	
	Time	Color	Time	Color
Stanford	8.696	115	98.152	113
dblp	49.876	255	339.137	120
youtube	27.792	167	172.035	45
RoadNet	30.438	110	183.646	6
Wiki	129.233	112	272.397	97
soc-lj	493.387	490	5002.41	329
RMAT	1892.932	82	7989.48	78
random	3431.787	89	12496.272	84
twitter	15319.55	1189	51823.1	910
webbase	10438.999	1650	186029.255	1507

Table 2 shows that recursion only coloring method can achieve better runtime performance as compared to sequential spread only coloring method, but it tend to use more colors on some datasets. As we explained this problem clearly in Section 2, this phenomenon occurs because that only active vertices updated in sequential spread only processing model, while all the vertices are updated in the recursion only coloring method. While the vertices are colored permanently once the color is choosed in recursion only coloring model, but the colors may changed in later iterations in recursion only coloring method. We can also conclude that random graph can be colored in fewer iterations under recursion only coloring method, because all the degrees of the vertices are change in a narrow space which can better suitable for GPU SIMD processing model.

5.4 Timing for Switching the Execution Stage

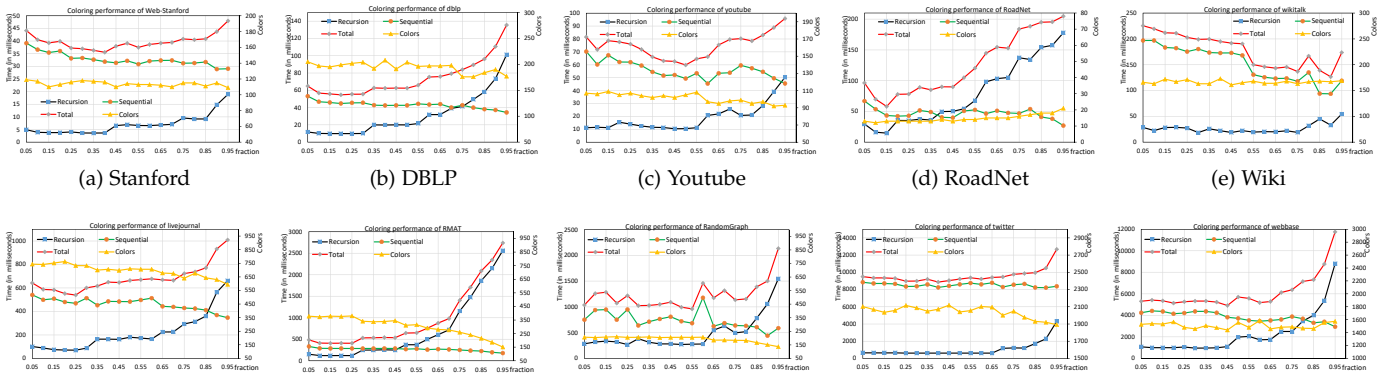
In order to find a suitable value of the parameter *fraction*, which is used to control when the execution of Feluca is switched from the recursion stage to the sequential spread stage, we designed a set of experiments for different datasets with different value of fractions. The execution time of these two stages in Feluca with different *fraction* values are shown in figure 5. The left side y-axis in figure 5 is the coloring time in milliseconds, while the y-axis at the right side is the number of colors. The red line with gray diamond dots shows the total coloring time of Feluca, while the black line and the green line show the coloring times of the recursion and the sequential spread stage, respectively. The number of colors is shown by the yellow line with triangle dots.

We can make the following observations from figure 5.

1) The sequential spread stage is most time consuming with a small *fraction* value, which means there are very few vertices colored in the recursion stage. Figure 5 shows that for all the power-law graphs, the execution time of the recursion stage is much smaller than that of the sequential spread stage with a small *fraction* value, which means the recursion algorithm is much faster than the sequential spread algorithm. On the contrary, the recursion method needs more time with a big *fraction* value, which means there are more conflicts occurred at the end of the recursion stage.
2) Feluca can achieve good performance on both power-law graphs and random graphs with a small number of colors.
3) Figure 5 shows that the execution time of Feluca is a convex function over *fraction*. Hence, Feluca can achieve the best coloring time when the derivative of the execution time function is close to 0. The derivative of the execution time function can be approximated as $\Delta t = \frac{t_i - t_{i-1}}{fraction_i - fraction_{i-1}}$. It can be assumed that the execution times of two consecutive iterations are almost the same. Then, Feluca can achieve minimal executing time once it switches from the recursion stage to the sequential spread stage when the number of active vertices is the same as the number of conflicting vertices in two consecutive iterations.

Since Feluca assigns the colors to the conflicting vertices following the edges directions, most conflicting vertices can find suitable colors in the first few iterations. However, after a majority of vertices find the suitable colors, these colored vertices will have impact on the colors of the remaining vertices. This causes a small number of remaining vertices to change their colors repeatedly in later iterations and therefore slows down the progress. This is why Feluca switches from the recursion stage to the sequential spread stage when the condition stated in the last observation made from figure 5 is met.

As discussed above, Feluca can achieve the minimal executing time if it switches from the recursion stage to the sequential spread stage when the number of active vertices is the same as the number of conflicting vertices in two consecutive iterations. As we revealed in this paper, the coloring time is a convex function over the switching time point, which indicates that as we move the switching time point earlier, the number of used colors decreases while the coloring time increases. Since Feluca is a graph coloring algorithm, whose primary goal should be using as few colors as possible. The balance between the number of used colors and the coloring time should tilt towards the former. Hence, we conducted the experiments to investigate the impact of moving the switching time point earlier on both the coloring time and the number of used colors. Table 3 shows the number of used colors and the coloring time for processing the ten datasets when the switch point is set as when the number of active vertices is more than the number of conflicting vertices in two consecutive iterations by α percent. Higher value of the parameter α, the earlier the switch time point is. From this table, we can observe that 1) indeed moving the switch time point early can reduce the number of used colors but increase the coloring time; 2) as we move the switch time point further earlier, the less number of colors can be reduced but the coloring time increases more prominently; and 3) when α is 10%, we can obtain a much less number of used colors without increasing the coloring time by too much.



(a) Stanford (b) DBLP (c) Youtube (d) RoadNet (e) Wiki

(f) soc-lj (g) RMAT (h) Random (i) Twitter (j) Webbase

Fig. 5: Coloring time with different *fraction*. X-axis is the value of *fraction*, the Y-axis on the left is the coloring time in milliseconds, while Y-axis on the right is the number of colors. "Sequential" means the time spent by the sequential spread stage of the coloring algorithm while "recursion" means the time by the recursion stage. The parameter *fraction* indicates the ratio of the number of colored vertices in the recursion stage to the number of total vertices in the graph.

5.5 Comparison Against the State-of-the-art Techniques

We compared Feluca with some state-of-the-art methods in this area, such as kokkos [20], Gunrock [36], GraphBLAST [41], ChenGC [42], [43], SIRG [44], cuSPARSE [40] and JPL [40]. In this experiment, Feluca switches the execution stage by setting α to 10%. Namely, Feluca switches from the recursion stage to the sequential spread stage when the number of active vertices is 10% more than the number of conflicting vertices in two consecutive iterations. Table 4 shows the execution time and the number of colors used for all ten graphs. A performance value plotted in each graph is the average of 5 independent runs of the GPU-based solutions (Feluca, kokkos, Gunrock, GraphBLAST, ChenGC, SIRG, cuSparse, and JPL).

The experimental results show that Feluca achieves up to 8.39×, 14.70×, 7.55×, and 9.70× speed up over kokkos [20], Gunrock [36], SIRG [44] and ChenGC [42], [43], respectively.

Table 4 shows that Feluca outperforms all other competitors in terms of run-time with all ten datasets. All these algorithms can generate a complete coloring plan except cuSPARSE, which is an approximate coloring algorithm. In cuSPARSE, the parameter *fractionToColor* is the *fraction* of nodes to be colored, which should be in [0.0, 1.0]. The algorithm stopped when the number of the colored vertices is the *fractionToColor* percentage of the whole vertices of the graph. In the above experiments, there are still so many vertices that are not colored correctly even when we set *fractionToColor* as 1.0. For example, out of the 4,847,571 vertices in `soc-lj`, 41,652,230 vertices in `twitter`, 118,142,155 vertices in `webbase`, cuSPARSE assigns 2,437,231 and 22,396,212 and 104,173,619 vertices, respectively, to the same color. From another point of view, cuSPARSE does not choose another right color for the vertices when the conflicts occur. 41,652,230 vertices of `twitter` are colored with 947 colors in Feluca while cuSPARSE only colored the 19,256,018 (46% of all the vertices) vertices using 917 colors and assigns the remaining 22,396,212 vertices to the same color. This is the main reason why cuSPARSE can achieve a fewer number of colors on `soc-lj`, `twitter` and `webbase`.

As described in section 4.1, Feluca only focuses on the current vertex and their parents and does not search the entire color array to find the available color for the current vertex. Although this scheme avoids the use of atomic operations (hence improve the run-time performance), it may increase the number of used colors to some extent. Table 4 shows kokkos can color the `Stanford` and `Wiki` datasets with the fewer colors than Feluca.

5.6 The Color-centric Scheme in Feluca

In this experiment, we implement Feluca with and without the color-centric paradigm. The experiment is designed to show the ratio of the number of conflicting vertices to the number of active vertices in each iteration. The experiment result is shown in figure 6. The figure shows that, without the color-centric paradigm the tested three datasets, `Stanford`, `youtube` and `random`, need at least 24 iterations to converge. While with the color-centric paradigm, `random` converged at the 7^{th} iteration and other two datasets converged at the 11^{th} iteration.

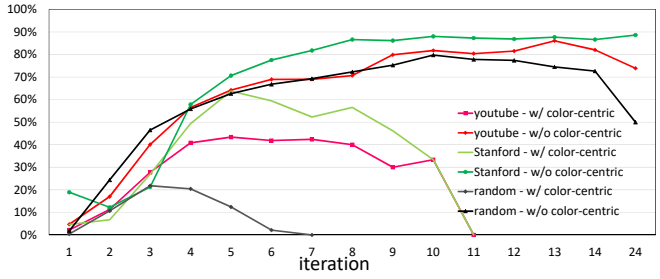


Fig. 6: The ratio of the number of conflicting vertices to the number of active vertices in each iteration with and without color-centric optimization.

Figure 6 also shows that with the color-centric paradigm, the ratio of the number of conflicting vertices to the number of active vertices in each iteration is no more than 45% for youtube and RandomGraph, while the conflict ratio can increase to 80% 87% without the color-centric paradigm.

The color-centric paradigm can avoid about 50% conflicts for these two datasets. The conflict ratio of `Stanford` increased to 87% at the 10^{th} iteration, while the conflict ratio is no more than 63% with the color-centric paradigm.

Figure 7 shows the percentage of issue slots that issued at least one instruction, averaged across all cycles. The figure shows that the color-centric paradigm can improve the performance by 111% (on `RoadNet` dataset) 410% (on `Stanford` dataset), which indicates that more instruction were executed in every iteration by using the color-centric paradigm.

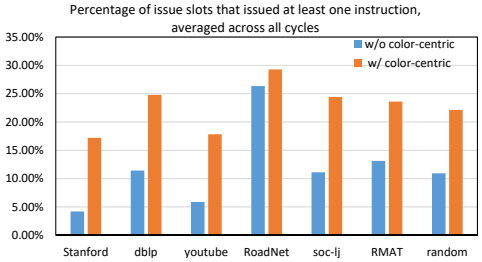


Fig. 7: The percentage of the issue slots that issued at least one instruction, averaged across all cycles, with and without the color-centric scheme.

Figure 8 plots the average number of warps that are eligible to issue per active cycle. It shows that on the `RoadNet` dataset, the average number of warps in each active cycle is no more than 2.2 without the color-centric paradigm, while with the color-centric paradigm it increases to 3.12. On `Stanford` dataset, the color-centric paradigm can improve the average number of warps in each active cycle by up to 4.9×. These experiments show that the color-centric paradigm can improve the active warps in each execution cycle, which means the there are more active threads by using the color-centric paradigm.

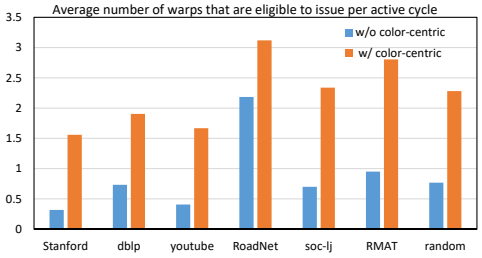


Fig. 8: The average number of warps that are eligible to issue per active cycle with and without color-centric optimization.

5.7 Running Feluca on Different GPU Devices

In order to show the performance of Feluca on three different GPU device, NVIDIA K20, NVIDIA K40 and NVIDIA P100. The configuration of NVIDIA K20 is the same as in previous experiments. There are 2880 CUDA cores and 12GB on-board memory in NVIDIA K40, and there are 3,584 CUDA cores and 16GB on-board memory in NVIDIA P100. Figure 9 shows the performance achieved by Feluca scales well with the increase of the GPU capability.

On the other hand, we also implement Feluca on a node equipped with multi-GPUs. In this implementation, we first partition the graphs into several blocks and then assign the blocks to GPUs in a round-robin fashion. A GPU needs to synchronize its coloring results with other GPUs to obtain the final coloring plan. In Feluca, we begin to color the graph block once it was loaded into the GPU, and we also begin to transfer the next graph block to a GPU immediately. We overlapped the computation with communication by using this round-robin graph transfer method. In the multi-GPU version, the synchronization is costly because fast tasks need to wait for slow tasks. Furthermore, the synchronization communication is completed through PCI-E. In order to test the scalability of Feluca on multi-GPUs, we run Feluca on a node that equipped with 2 NVIDIA K20m and 2 NVIDIA P100 GPUs, the experiments show that Feluca can achieve 2.57 – 5.16× speedup on a multi-GPU platform over a single NVIDIA K20 GPU. From this experiment, we can conclude that the communication cost for multi-GPU coloring is much higher than the computation due to the synchronization among multiple GPUs. We will try to design a scalable graph coloring algorithm for multi-GPUs in our further work.

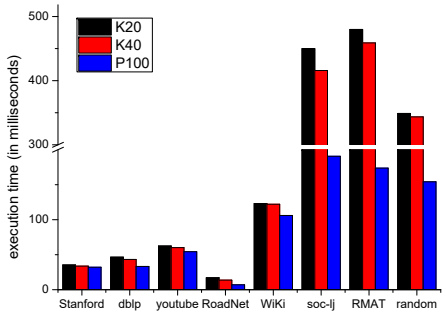


Fig. 9: The performance of Feluca on different GPU devices.

6 Related Work

Although the graph coloring is an NP-Hard problem, it has been included in the DIMACS Implementation Challenge [1] and there are vast amount of research work have been done. This section introduce the recent state-of-the-art works on parallel graph coloring.

The existing research shows that the simple greedy implementation can archive near optimal solutions [45]. However, it is inherently sequential, which is difficult to be parallelized. Gebremedhin and Manne [46] proposed a multi-threaded parallel algorithm based on the greedy coloring scheme. In their work, all the threads run as a greedy coloring algorithm asynchronously in the first phase. There exist many conflicts in this execution model, due to the reason that two threads may color two neighboring vertices at the same time. The authors check the conflicts in the second phase and store the conflict vertices in a table. In the third phase, the conflict vertices are colored sequentially. The iterative coloring approach was first proposed based on the independent set finding method [25], and later this

1. http://archive.dimacs.rutgers.edu/Challenges/

coloring scheme is adopted on the shared-memory systems [47]. The proposed scheme is easy to be parallelized. But the atomic operation is needed for the conflict resolution. On the other hand, this method may also cause more colors being needed. Unlike the previous work, we propose a color-centric paradigm in this work to improve the degree of parallelism for the sequential spread stage. In our color-centric paradigm, we organize the GPU thread blocks in pipeline. There is no conflict occurring with this paradigm.

Another important aspect of the parallel graph coloring algorithm is conflict resolution. In the work by Deveci et al., the atomic operation is used to resolve the conflicts [20]. Following Gebremedhin and Manne's work [46], Rokos et al. proposed a variation, which combined the conflict detection and resolution phase together, to reduce the number of kernel launches [48]. Bozda et al. proposed a parallel graph coloring algorithm based on the first-fit method [49]. In this work, each processor colors its assigned vertices by the chosen colors from a section of the whole color array. This method reduces the conflicts to some extents. However, it needs a temporary array to locate the candidate color area for a thread, which leads to more memory consumption. This color chosen scheme may also cause a large number of colors to be used. A hybrid MPI and OpenMP implementation was developed by Sariyce et al. [50]. Similar to the work by Deveci et al. [20], the atomic operation is used to resolve the conflicts. In our work, we propose a top-down coloring scheme, which instructs every thread to chose a suitable color for the current active vertex according to its parents color. In our proposed method, a fixed-length temporary array, which holds the colors that were just used, is used to locate the candidate color area from the color array, which can avoid most of the conflicts. On the other hand, the memory size of the temporary array is very small. For example, two integers are enough.

There are also a few works focusing on the GPU optimization. Çatalyürek et al. proposed a multi-threaded dataflow algorithm for Cray XMT [47], which replies on the hardware support and it is not suitable for the current GPU. Grosset et al. implemented the G-M algorithm [51] on GPU [21]. But the authors left part of the conflict resolution work to CPU. Naumov et al. developed a fast coloring heuristic [40], which is part of the widely used cuSPARSE library. This algorithm is a variation of Jones and Plassmann (JP) [52] independent set based coloring algorithm. It is fast, but usually needs more colors than JP. Che et al. studied the variations of the JP algorithm on GPU [53], they implemented the work-stealing technique and designed a hybrid algorithm to address the load imbalance problem. In our work, we implemented the edge-list graph representation method and added some virtual edges to address the load imbalance problem, which do not consume any memory space.

Gunrock [36] is a data-centric graph processing library, the authors abstract the frontier operation into the steps of advance, filter and compute, rather than the widely used operations of Gather, Apply and Scatter. Combined with the load balancing and workload management techniques, Gunrock can achieve the overall system performance comparable to some other graph processing frameworks. GraphBLAST [41] is a sparse linear algebra based graph processing library on GPU. By implementing load balancing, memory efficiency, and some general optimization techniques (push/pull), GraphBLAST provides an easy-to-use interface and competitive performance. There is extension work on Gunrock and GraphBLAST to implement graph coloring algorithms.

Osama at al. [54] map the graph coloring algorithm based on the independent set to Gunrock and GraphBLAS by using the JPL heuristic. SIRG [44] proposes a re-coloring scheme, by using a data-driven programming method, to handle the conflict vertices. Chen et al. [42], [43] assign the threads by using a work-efficient manner, which is similar with Kokkos [20]. But the authors use a greedy algorithm to assign the colors. Their implementation can improve the utilization of threads. On the other hand, however, the greedy color assignment limits the parallelization of the algorithm. In our work, we combine the recursion-based method with the sequential spread-based method. Namely, both the greedy approach and the approach based on the independent set are used to process the vertices at different execution stages, which can improve the thread efficiency and maintain the parallelization of the algorithm.

7 Conclusion and Future Opportunities

We present Feluca, a highly efficient hybrid GPU graph coloring algorithm by adaptively switching execution model based on monitoring the conflict rate of the active vertices in each iteration. This approach is necessary to acquire the advantage of both sequential spread and recursion based coloring method, and consequently and significantly improve overall performance. Furthermore, the craftly-designed color-centric coloring paradigm, which improved the degree of parallelism for the sequential spread part. Our intensive experiments show the high effectiveness of Feluca.

In the future, we plan to focus on other aspects of graph coloring, such as coloring the graphs in hybrid systems, coloring the graphs on new devices and coloring the dynamic graph on GPU, multi-GPU computer, and other new devices.

Acknowledgments

This work is partly supported by National Key R&D Program of China (No. 2017YFC0803700), the National Science Foundation of China (No. 61772218).

References

[1] D. Marx, "Graph colouring problems and their applications in scheduling," *Periodica Polytechnica Electrical Engineering*, vol. 48, no. 1-2, pp. 11–16, 2004.

[2] G. J. Chaitin, "Register allocation and spilling via graph coloring," Feb. 1986, uS Patent 4,571,678.

[3] A. M. Herzberg and M. R. Murty, "Sudoku squares and chromatic polynomials," *Notices of the American Mathematical Society*, vol. 54, pp. 708–717, June/July 2007. [Online]. Available: http://www.ams.org/notices/200706/

[4] G. Chartrand and P. Zhang, *Introduction to Graph Theory*. McGraw Hill Education, 2017.

[5] M. R. Garey and D. S. Johnson, "The complexity of near-optimal graph coloring," *Journal of the ACM*, vol. 23, no. 1, pp. 43–49, Jan. 1976. [Online]. Available: http://doi.acm.org/10.1145/321921.321926

[6] M. K. Ta, K. Kaya, and E. Saule, "Greed is good: Parallel algorithms for bipartite-graph partial coloring on multicore architectures," in *Proceedings of 2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 503–512.

[7] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 3143. [Online]. Available: https://doi.org/10.1145/3018743.3018755

[8] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 201213. [Online]. Available: https://doi.org/10.1145/3293883.3295716

[9] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 233–245.

[10] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren, "Graphphi: Efficient parallel graph processing on emerging throughput-oriented architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243205

[11] D. Mawhirter and B. Wu, "Automine: Harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 509523. [Online]. Available: https://doi.org/10.1145/3341301.3359633

[12] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," *SIGPLAN Not.*, vol. 53, no. 2, p. 622636, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173180

[13] H. Park and M.-S. Kim, "Evograph: An effective and efficient graph upscaling method for preserving graph properties," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 20512059. [Online]. Available: https://doi.org/10.1145/3219819.3220123

[14] G. Chen, X. Shen, B. Wu, and D. Li, "Optimizing data placement on gpu memory: A portable approach," *IEEE Transactions on Computers*, vol. 66, no. 3, pp. 473–487, 2017.

[15] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross, "On-the-fly workload partitioning for integrated cpu/gpu architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243210

[16] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 14. IEEE Press, 2014, p. 719730. [Online]. Available: https://doi.org/10.1109/SC.2014.64

[17] J. Maglalang, S. Krishnamoorthy, and K. Agrawal, "Locality-aware dynamic task graph scheduling," in *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 70–80.

[18] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 297311. [Online]. Available: https://doi.org/10.1145/3037697.3037709

[19] M. E. Belviranli and J. S. Vetter, "Flame: Graph-based hardware representations for rapid and precise performance modeling," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1775–1780.

[20] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS' 16, May 2016, pp. 892–901.

[21] A. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on gpus," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP' 11, 2011, pp. 297–298.

[22] M. Kubale, *Graph Colorings*, ser. Contemporary Mathematics. American Mathematical Society, 2004.

[23] T. Husfeldt, *Topics in Chromatic Graph Theory*, ser. Graph Colouring Algorithms, L. W. Beineke and R. J. Wilson, Eds. Cambridge University Press, 2015.

[24] A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. Chavarria-Miranda, and S. Krishnamoorthy, "Approximate computing techniques for iterative graph algorithms," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 23–32.

[25] Çatalyürek Ümit V., J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 10, pp. 576 – 594, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819112000592

[26] L. Addario-Berry, S. Bhamidi, S. Bubeck, L. Devroye, G. Lugosi, and R. I. Oliveira, "Exceptional rotations of random graphs: A vc theory," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1893–1922, Jan. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789272.2886810

[27] S. Arora and E. Chlamtac, "New approximation guarantee for chromatic number," in *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '06. New York, NY, USA: ACM, 2006, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/1132516.1132548

[28] A. Rok and B. Walczak, "Outerstring graphs are χ-bounded," in *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, ser. SOCG'14. New York, NY, USA: ACM, 2014, pp. 136–143. [Online]. Available: http://doi.acm.org/10.1145/2582112.2582115

[29] K. Kothapalli and S. Pemmaraju, "Distributed graph coloring in a few rounds," in *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '11. New York, NY, USA: ACM, 2011, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1993806.1993812

[30] L. Barenboim and M. Elkin, "Deterministic distributed vertex coloring in polylogarithmic time," *Journal of the ACM*, vol. 58, no. 5, pp. 1–25, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2027216.2027221

[31] L. Barenboim, M. Elkin, and T. Maimon, "Deterministic distributed (delta + o(delta))-edge-coloring, and vertex-coloring of graphs with bounded diversity," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087812

[32] M. Ghaffari, J. Hirvonen, F. Kuhn, and Y. Maus, "Improved distributed delta-coloring," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, ser. PODC '18. New York, NY, USA: ACM, 2018, pp. 427–436. [Online]. Available: http://doi.acm.org/10.1145/3212734.3212764

[33] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 345–354. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370866

[34] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600227

[35] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin, "Frog: Asynchronous graph processing on gpu with hybrid coloring model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 1, pp. 29–42, Jan 2018.

[36] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP' 15. New York, NY, USA: ACM, 2015, pp. 265–266. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688538

[37] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, ser. PACT '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 39–50. [Online]. Available: https://doi.org/10.1109/PACT.2015.15
[38] C. S. D. of the University of Milan, "Laboratory for web algorithmics," http://law.di.unimi.it/index.php, November 2002, accessed July 17, 2018.
[39] J. Leskovec, "Stanford network analysis project," http://snap.stanford.edu/index.html, November 2009, accessed July 11, 2018.
[40] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the gpu," NVIDIA, Technical Report NVR-2015-001, May 2015.
[41] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," 2019.
[42] X. Chen, P. Li, J. Fang, T. Tang, Z. Wang, and C. Yang, "Efficient and high-quality sparse graph coloring on gpus," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4064, 2017, e4064 cpe.4064. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4064
[43] P. Li, X. Chen, Z. Quan, J. Fang, H. Su, T. Tang, and C. Yang, "High performance parallel graph coloring on gpgpus," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 845–854.
[44] M. A. Sistla and V. K. Nandivada, "Graph coloring using gpus," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed. Cham: Springer International Publishing, 2019, pp. 377–390.
[45] T. F. Coleman and J. J. Moré, "Estimation of sparse hessian matrices and graph coloring problems," *Mathematical Programming*, vol. 28, no. 3, pp. 243–270, Oct 1984. [Online]. Available: https://doi.org/10.1007/BF02612334
[46] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency and Computation: Practice and Experience*, vol. 12, no. 12, pp. 1131 – 1146, 2000. [Online]. Available: https://doi.org/10.1002/1096-9128(200010)12:12⟨1131::AID-CPE528⟩3.0.CO;2-2
[47] E. Saule and Çatalyürek Ümit V., "An early evaluation of the scalability of graph algorithms on the intel mic architecture," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 1629–1639.
[48] G. Rokos, G. Gorman, and P. H. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Proceedings of the 2015 European Conference on Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 414–425.
[49] B. Doruk, A. H. Gebremedhin, F. Manne, E. G. Boman, and Çatalyürek Ümit V., "A framework for scalable greedy coloring on distributed-memory parallel computers," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515 – 535, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373150700144X
[50] A. E. Sariyüce, E. Saule, and U. V. Çatalyürek, "Scalable hybrid implementation of graph coloring using mpi and openmp," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 1744–1753.
[51] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," *Theory of Computing*, vol. 3, no. 6, pp. 103–128, 2007. [Online]. Available: http://www.theoryofcomputing.org/articles/v003a006
[52] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, May 1993. [Online]. Available: http://dx.doi.org/10.1137/0914041
[53] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the gpu and some techniques to improve load imbalance," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 610–617.
[54] M. Osama, M. Truong, C. Yang, A. Bulu, and J. Owens, "Graph coloring on the gpu," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 231–240.

TABLE 3: Performance of Feluca with different switching point

Datasets	$\alpha = 0$		$\alpha = 5\%$		$\alpha = 10\%$		$\alpha = 15\%$	
	Time	Color	Time	Color	Time	Color	Time	Color
Stanford	28.810	92	36.623	76	42.690	64	68.372	53
dblp	42.388	139	46.737	126	54.555	114	62.566	106
youtube	58.777	57	72.401	48	86.151	39	93.363	37
RoadNet	12.524	6	69.904	6	46.804	5	69.904	5
Wiki	122.464	95	169.683	91	190.811	80	224.433	76
soc-lj	448.574	427	587.678	382	1180.120	325	9547.690	279
RMAT	410.309	373	876.434	281	1811.190	79	2740.070	72
random	283.491	207	320.409	137	353.643	86	640.600	68
twitter	2275.830	2068	4298.45	1892	5040.660	947	8847.250	873
webbase	1066.990	1575	2044.13	1564	2729.500	1559	4416.150	1537

TABLE 4: Running performance of different coloring algorithms

Algorithm		Stanford	dblp	youtube	RoadNet	Wiki	soc-lj	RMAT	random	twitter	webbase	Speedup
Feluca	time	42.690	54.555	86.151	46.804	190.811	1180.120	1811.190	353.643	5040.660	2729.5	–
	color	64	114	39	5	80	325	79	86	947	1559	
kokkos	time	50.765	183.034	230.806	162.546	631.069	1629.902	4454.264	2968.788	null	null	1.19 – 8.39
	color	45	119	46	6	65	330	87	94	null	null	
kokkos_MIC	time	2200.720	6106.440	3022.930	4893.600	5042.670	63284.500	163517.000	96557.300	728099	940540	26.42 – 344.58
	color	45	119	46	5	67	251	93	103	679	1226	
Gunrock	time	200.456	104.335	214.082	277.802	2804.297	null	4842.506	3613.233	5288.629	null	1.05 – 14.70
	color	57	136	41	8	155	null	134	163	1013	null	
GraphBLAST	time	71.005	309.587	331.184	682.126	745.667	4518.840	11127.100	11783.700	23026.600	60001.9	1.66 – 33.32
	color	78	126	76	6	199	330	121	128	1517	1363	
ChenGC	time	135.323	435.936	385.500	453.840	645.778	9474.651	3728.278	2124.772	7589.630	null	1.51 – 9.70
	color	240	153	194	32	394	557	210	203	1639	null	
SIRG	time	100.230	172.978	86.780	246.134	448.208	2612.175	2265.292	2669.664	7549.797	null	1.25 – 7.55
	color	65	119	68	6	135	352	160	157	960	null	
JPL	time	1121.790	787.762	1409.490	806.787	8703.570	11000.500	18774.900	17264.300	null	null	9.32 – 48.82
	color	169	121	262	13	489	646	316	334	null	null	
cuSPARSE	time	1463.250	541.429	617.205	531.205	818.762	1792.390	6910.610	9264.060	41201.300	264022	1.52 – 96.73
	color	95	70	103	32	159	184	129	112	917	412	
	Incorrect	134993	933161	283515	1953477	184167	2437231	6966819	4804572	22396212	104173619	

Note: Null means that the algorithm cannot process such a dataset. The `kokkos_MIC` is run on the Intel(R) Xeon(R) E5-2670 CPU with 16 threads, while others are run on NVIDIA Tesla P100 GPU. Reference [20] shows that kokkos can achieve the best performance by using the edge-based coloring method. So we set the parameter `--algorithm` as COLORING_EB for kokkos and `kokkos_MIC` in our experiments. The execution time is in milliseconds.