Efficient Algorithms for Graph Coloring on GPU

1st Nguyen Quang Anh, Pham
School of Computer Science and Engineering
Nanyang Technological University
Singapore
nqapham@ntu.edu.sg

2nd Rui Fan
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
fanrui@shanghaitech.edu.cn

***Abstract*—Graph coloring is an important problem in computer science and engineering with numerous applications. As the size of data increases today, graphs with millions of nodes are becoming commonplace. Parallel graph coloring algorithms on high throughput graphics processing units (GPUs) have recently been proposed to color such large graphs efficiently. We present two new graph coloring algorithms for GPUs which improve upon existing algorithms both in coloring speed and quality. The first algorithm, *counting-based Jones-Plassmann (CJP)*, uses counters to implement the classic Jones-Plassmann parallel coloring heuristic in a work-efficient manner. The second algorithm, *conflict coloring (CC)* achieves higher parallelism than CJP, and is based on optimistically coloring the graph using estimates of the chromatic number. We compared CC and CJP with two state-of-the-art GPU coloring algorithms, csrcolor [1] and Deveci *et al*'s [2] vertex/edge-based algorithms (which we call VEB), as well as the sequential CPU algorithm ColPack [3]. In terms of coloring quality, CJP and CC are both far better than csrcolor, while CJP uses 10% fewer colors than VEB on average and CC uses 10% more. Compared to ColPack, CJP and CC use $1.3\times$ and $1.5\times$ more colors on nonbipartite graphs, resp. In terms of speed, CJP is on average $1.5-2\times$ faster than the other algorithms, while CC is $2.7-4.3\times$ faster.**

***Index Terms*—graph coloring, parallel, GPU, high performance, randomized algorithm**

I. Introduction

In the graph coloring problem, we assign colors to the nodes of a graph so that neighboring nodes have different colors, while trying to minimize the number of colors used. In practice, nodes often represent tasks and edges conflicts between tasks. Coloring the graph partitions the tasks into a small number of non-conflicting classes, each of which can be performed in parallel. Graph coloring has a number of applications in computer science and engineering, including processor and network scheduling [4] , sparse linear algebra [5] , matrix preconditioning [6] , optimization [7], community detection [8] and computational geometry [9].

Recent years have seen large growth in the size of data, and applications now frequently process graphs with millions of nodes or more. To color these large graphs efficiently, a number of parallel graph coloring algorithms have been developed for shared and distributed memory systems [10], [11], [12], [13]. More recently, coloring algorithms have also been proposed for graphics processing units (GPUs). These include the algorithm by Cohen and Castonguay [14], which is implemented as the *csrcolor* function in Nvidia's cuSPARSE library [1], and Deveci *et al*'s [2] recent *vertex-based* and *edge-based* coloring algorithms, which we refer to as VEB. GPUs are an attractive platform for coloring, both due to their high throughput and also because of their wide use in linear algebra and optimization, two of the main application areas for graph coloring. In this paper, we present two new GPU coloring algorithms, *counting-based Jones-Plassmann (CJP)* and *conflict coloring (CC)*. CJP is optimized for coloring quality, and is better for applications where minimizing color usage is most important. CC is optimized for speed, and is more suitable for applications where coloring takes a large proportion of the overall running time, or where colorings need to be computed repeatedly.

CJP is based on the classic Jones-Plassmann parallel coloring algorithm [10], which operates in rounds and colors a maximal independent set of nodes in each round. We propose a work-efficient implementation of the algorithm, by associating each node with a counter value to quickly identify the round in which the node should be colored. This allows CJP to examine each edge in the graph only a constant number of times during an execution. To reduce color use, CJP colors each node using the minimum available color. We show how to perform this operation efficiently on the GPU. In contrast, Naumov et al. [1] color nodes using a multiple of the round number plus an offset, resulting in much higher color use.

CC further improves on the speed of CJP, at the expense of using somewhat more colors. Whereas CJP always ensures nodes have a valid coloring at any stage of execution, CC allows neighboring nodes to temporarily have the same color. These coloring conflicts are later detected and some of the conflicting nodes are recolored in the next round. While this results in somewhat more work than CJP, it also allows greater parallelism and better utilization of the GPU's massive multithreading. In contrast, the parallelism in CJP is limited by the size of the independent set in each round. In CC, nodes are assigned random colors, up to a current estimate of the chromatic number (i.e. the minimum number of colors needed to color the graph). We show how nodes can independently update the estimate to quickly converge on an accurate value. Choosing a random color instead of the minimum one results in fewer conflicts and faster termination compared to other speculative coloring algorithms [11], [12], [13].

We compared CJP and CC to several other coloring algorithms, including the GPU-based algorithms csrcolor and

978-1-5386-7308-9/18/$31.00 ©2018 IEEE
DOI 10.1109/ICPADS.2018.00066

VEB, and the CPU-based sequential coloring algorithm in ColPack [3]. Sequential algorithms generally use fewer colors than parallel ones, since they avoid certain race conditions and can effectively implement several color-reducing node ordering heuristics. Thus, the comparison to ColPack provides a baseline to assess CJP and CC's coloring quality. We performed experiments using a variety of graphs from the University of Florida sparse matrix collection [15]. We found that CJP was on average $1.5\times, 2\times$ and $1.9\times$ faster than csrcolor, VEB and ColPack resp., while CC was on average $4.3\times, 3.3\times$ and $2.7\times$ faster, resp. In terms of coloring quality, ColPack almost always provided the best results. CJP and CC were both much better than csrcolor, and CJP was somewhat better than VEB while CC was somewhat worse. For nonbipartite graphs, CJP used on average $0.23\times, 0.9\times$ and $1.3\times$ as many colors as csrcolor, VEB and ColPack resp., while CC used $0.28\times, 1.1\times$ and $1.5\times$ as many colors resp. Lastly, we examined CJP and CC in terms of execution efficiency. We found that CC colors more nodes each round, and generally terminated in 5 to 7 rounds while CJP took 20 to 35 rounds. In addition, CC's conflict reduction mechanism was quite successful, and each node was processed on average only 1.3 times in CC.

The rest of the paper is organized as follows. In Section II, we describe related work on parallel graph coloring and also give some background on GPU programming. We present the CJP and CC algorithms and give the main techniques in Section III. In Section IV we show our experimental results and discuss various performance properties. Finally, we conclude in Section V.

II. Background

A. Related work

Graph coloring is a classic problem in computer science and has been studied extensively in both the theoretical and applied domains. Coloring a graph with the minimum number of colors is NP-hard [16], but a number of heuristics exist to find good colorings quickly. The *first fit* greedy method visits the graph in an arbitrary order and colors each node using the smallest color not used by its neighbors. The *largest degree first* heuristic [17] visits nodes in order of nonincreasing degree and has been shown to reduce the number of colors used. Other effective node orderings include *smallest degree last* [18], *incidence degree* [19] and *saturation degree* [20]. Gebremedhin *et al* presented efficient implementations of these heuristics and others in the CPU-based graph coloring library ColPack [3]. Allwright et al. [21] implemented and compared performance of parallel versions of *largest degree first* and *smallest degree last* heuristics. Hasenplaugh *et al* [22] recently introduced several orderings for parallel graph coloring with provable performance guarantees and implemented them on multicore CPUs.

One of the first works on parallel graph coloring was by Luby *et al* [23], and used a round based scheme which colored a maximal independent set of nodes in each round. A random number is assigned to each node, and each independent set is formed by choosing all the uncolored nodes whose values are larger than those of their uncolored neighbors. Jones and Plassmann [10] extended this algorithm to the asynchronous setting. Our CJP algorithm is based on the Jones-Plassmann technique optimized for GPUs.

Gebremedhin and Manne [11] proposed an optimistic algorithm for coloring graphs in parallel on shared memory computers. Their technique consists of three phases. Each processor is responsible for a set of nodes, and in the first phase the processor colors its own nodes, possibly resulting in conflicting colorings with other processors. In phase 2 processors detect conflicts in parallel, and finally in phase 3 the conflicting nodes are recolored sequentially. Çatalyürek *et al* [12] extended this approach by adding the conflicting nodes to a list and recursively coloring them in later rounds. Rokos *et al* [13] further improved the performance of [12] by reducing the amount of synchronization.

Our CC algorithm is similar to the methods in [11] and [12] in that we also allow nodes to have conflicting colors and later recolor them. However, a key difference is that we choose a random unused color within some range for each node, whereas [11] and [12] choose the minimum allowed color. Using a random color gives each node a larger range of colors to select from, which reduces the number of conflicts and the amount of redundant work. The difference is especially important on GPUs. Indeed, Rokos *et al* [13] state that their implementation of [12]'s algorithm on GPUs failed to terminate due to an infinite loop caused by choosing the minimum color. The difficulty in using a random color is choosing the range to select the color from. In CC we propose a method to dynamically adjust the color range so that it quickly attains a size large enough for avoiding conflicts while still minimizing color use.

In recent years GPUs have emerged as an important platform for high performance and general purpose computing. Grosset *et al* adapted [11]'s algorithm to the GPU but perform some computations on the CPU. Cohen and Castonguay [14] introduced an algorithm based on the Luby-Jones-Plassmann (LJP) technique, and which is implemented as the csrcolor function in Nvidia's popular cuSPARSE library. They introduced an optimization which assigns multiple random values to each node, which improves coloring speed at the expense of using substantially more colors. Recently Naumov *et al* [1] applied Cohen's algorithm [14] to perform efficient incomplete LU factorization on the GPU, improving performance up to 8 times compared to the CPU. Che *et al* [24] proposed a hybrid coloring method for the GPU which improves load balancing. In early stages of the algorithm they color high degree nodes as in the largest degree first heuristic, while in later stages they use the LJP technique. Deveci *et al* [2] recently introduced two high quality coloring algorithms based on Gebremedhin and Manne's optimistic technique [11] for the GPU and Xeon Phi processors. Their vertex-based algorithm processes an entire node using a thread while the edge-based algorithm processes one edge per thread. The edge-based technique generally uses fewer colors and is also faster. However, each edge may be processed multiple times and so the total amount of work done by the algorithm may be high.

For more details about the vast literature on graph coloring we refer the reader to the survey by Gebremedhin *et al* [25]. In addition to basic graph coloring, the survey also discusses problems such as distance-k, star and hypergraph coloring, and also discusses applications for these algorithms and additional theory. Other works on bounds for the chromatic number can be found in [26], [27], [28] and [29], among many others.

III. Coloring Algorithms

In this section we present the CJP and CC coloring algorithms. We give the main ideas for the algorithms in Sections III-A and III-B, then discuss how to efficiently implement them on GPUs in the next section.

A. Counting-based Jones-Plassmann algorithm

Our first algorithm *CJP* is an efficient implementation the Jones-Plassmann technique. CJP associates a counter with each node to quickly determine the round in which a node should be colored. This allows the algorithm to be work efficient, and examine each edge only a constant number of times during an execution. Consider an undirected input graph $G = (V, E)$ with n nodes and m edges. The CJP algorithm performs the following steps.

1) Initially every node v is assigned a random value $val(v)$. Then v counts the number of neighboring nodes with higher values and stores the result in $count(v)$, i.e. $count(v) = |\{w \mid ((v, w) \in E) \wedge (val(w) > val(v)\}|$. Assume for simplicity all nodes have different values; if ties occur they can be broken by node ID. In general, $count(v)$ represents the number of v's neighbors that have higher values and that are still uncolored.
2) After a node v computes $count(v)$, it enqueues itself into a queue Q_1 if $count(v) = 0$. Q_1 is stored in global memory, and contains the first set of nodes to be colored. We insert nodes in parallel into Q_1 using the prefix sum technique in [30].
3) We then begin the main loop for the algorithm, which occurs in a series of rounds. Suppose we are currently in round $i \geq 1$. We will assign colors to the nodes in queue Q_i, which contains all the uncolored nodes that have higher values than all their uncolored neighbors. In addition, we will construct Q_{i+1}, the nodes to be colored in the next round.
   Each node v in Q_i first finds the minimum color not used by any of its neighbors and sets its own $color(v)$ to this value.
   Next, v iterates through all its neighbors and decrements $count(w)$ for any neighbor w with $count(w) > 0$. This is done because $val(v) > val(w)$, and so after coloring v node w will have one less uncolored neighbor with higher value. After the decrement, v checks if $count(w) = 0$. If so, then w's value is now higher than all those of its uncolored neighbors, so v enqueues w into Q_{i+1}.
4) Round i ends after all nodes in Q_i have been processed in step 3. Then if Q_{i+1} is not empty, we proceed to round $i + 1$ and repeat step 3. Otherwise, Q_{i+1} is empty and all the nodes in G have been colored.
   After finishing round i, Q_i is no longer needed. We can conserve memory by reusing Q_i's space to store queue Q_{i+2}, so that at all times we only need space for two queues.

1) Analysis of the CJP algorithm: We now show that CJP computes a correct coloring. We also show it runs in $O(m)$, where m is the number of edges. To show correctness, we first prove that at any point after step 2 of the algorithm, the *count* value of any node is at least as large as the number of uncolored neighbors of the node that have higher values.

Lemma 1. *Consider an execution of the CJP algorithm and a point in time t after step 2 in the execution. For any node v,* $count(v) \geq |\{w \mid ((v, w) \in E) \wedge (val(w) > val(v)) \wedge (w \textit{ is uncolored at } t)\}|$.

Proof. The proof is by induction. The lemma is true immediately after finishing step 2 of the execution. Assume it holds at time $t - 1$ (i.e. after $t - 1$ operations have been performed in the execution). Since a neighbor of v can never go from colored to uncolored, the quantity on the right hand side of the inequality in the lemma never increases. So it suffices to check the lemma still holds if the operation at time t decreases $count(v)$.

Since $count(v)$ decreases, v is currently uncolored. In addition, there must be a neighbor w of v in the queue at time $t - 1$ that decremented $count(v)$. Since w is in the queue, it has $count(w) = 0$. Then by the inductive hypothesis, w does not have any uncolored neighbors with higher values at time $t - 1$. So since v is an uncolored neighbor of w, v's value is less than w's. Before decrementing $count[v]$, w colored itself, which decreased the number of uncolored neighbors of v with higher values. Thus, after w decrements $count(v)$, the lemma still holds. □

Theorem 1. *The CJP algorithm produces a valid coloring.*

Proof. We show that at any time when two neighboring nodes v and w are colored, $color(v) \neq color(w)$. Suppose first that v and w are colored in different rounds, and WLOG v was colored first. Then when w chooses its color, it will read $color(v)$ and choose a different color. So since nodes never change their colors, we have $color(v) \neq color(w)$ at all future times. Next, suppose v and w are colored in the same round. Then they are both in the queue at some time t, and $count(v) = count(w) = 0$. Also, v and w are both uncolored at time t. Assume WLOG that $val(w) > val(v)$. We now have a contradiction, because Lemma 1 implies $count(v) \geq 1$, since w is an uncolored neighbor of v with higher value. Thus, v and w cannot be colored in the same round, and so the theorem holds. □

Next, we show the algorithm is work efficient.

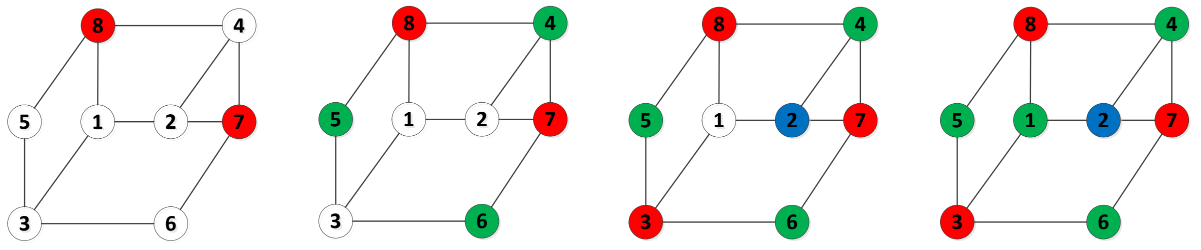


Fig. 1. Illustration of CJP. The value in each node is its hash value. **(a)** Two local maxima are colored using the first color. **(b)** Three additional nodes are colored using the second color. Note that these nodes are neighbors of nodes colored in step (a) and they become local maxima in this step. **(c)** Nodes 2 and 3 are colored. While node 2 is colored by a new color, node 3 is colored by the first color, which is the smallest available color for it. **(d)** Node 1 is colored. As its neighbors are colored by first and third color, it uses the second color.

Theorem 2. *Given a input graph with m edges, CJP runs in $O(m)$.*

Proof. We first show how to associate every piece of work done in the algorithm with an edge in the graph. The first type of work in the algorithm is to decrease $count(v)$ for some node v. When this happens, we can associate it with an edge (v, w), where w is a neighbor of v that was in the queue and caused the decrement. Also, for each edge (v, w), this type of work occurs at most once, since w is only in the queue for one round. The second type of work in the algorithm is for a node v to find a minimum unused color among its neighbors. This involves examining every neighbor w of v, and so we can associate the examination of w with edge (v, w). This work also only occurs once, since it happens when v is colored, which only happens once. The last type of work in the algorithm is to enqueue nodes. When a node v is enqueued, we can associate this with a neighbor w whose decrement of $count(v)$ caused v to be enqueued. The work only occurs once, since v is only enqueued once. Combining the above, we have that each piece of work in the algorithm is associated with some edge. Also, each edge is only associated with a constant number of pieces of work. Thus, the theorem follows. □

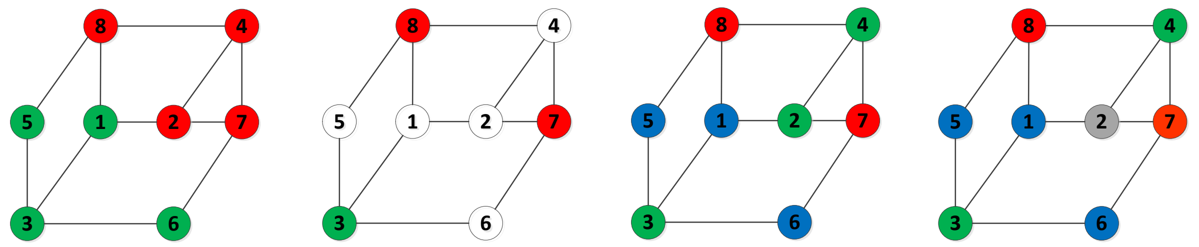
B. Conflict coloring algorithm

We now describe a second algorithm called *conflict coloring*, or *CC*. Like CJP, the CC algorithm also runs in rounds. However, whereas CJP maintains a valid coloring at all points in an execution, CC allows neighboring nodes to temporarily have the same color during a round. At the end of the round CC performs a conflict detection procedure that restores a valid coloring.

CC is significantly faster than CJP, with a key reason being that it has more parallelism. The amount of work in each round of CJP is determined by the size of the maximal independent set formed by nodes with $count = 0$. For some graphs this is much less than the tens of thousands of concurrent threads needed to fully occupy a GPU. CC on the other hand tries to color all uncolored nodes each round, resulting in more work. CC however uses slightly more colors than CJP on average. This is because nodes in CC choose a random color within some range, as opposed to choosing the minimum allowed color as in CJP. We discuss the performance and coloring quality of CC and CJP in more detail in Section IV-B.

The CC algorithm maintains a global variable $max\text{-}color$ giving the maximum color any node can use at the current time. $max\text{-}color$ represents an estimate of the graph's chromatic number. We initialize $max\text{-}color$ to 4, though this choice is somewhat arbitrary. During the course of an execution threads can increase $max\text{-}color$ when they find it is insufficient to color certain nodes; however, $max\text{-}color$ never decreases. The algorithm also maintains a queue Q of *active* nodes which still need to be colored. Initially Q contains all the nodes. Nodes not in Q are *inactive*, and these all have a color, which will never change. Lastly, each node v has a random value $val(v)$ used for tie-breaking, as we describe later. We now describe the main steps of the algorithm.

1) At the beginning of each round every node in Q chooses a random color from 1 to $max\text{-}color$ that is unused by its neighbors, at the time that it reads its neighbors' colors. Note that due to race conditions two nodes may choose the same color. We intentionally allow this situation in order to maximize parallelism.
   If v finds that all the colors from 1 to $max\text{-}color$ are used by its neighbors, then it atomically increments $max\text{-}color$, and selects the new value as its color. Note that in a round $max\text{-}color$ can increase multiple times. As we show in Section IV-B2, $max\text{-}color$ quickly grows large enough so that nodes picking random unused colors will rarely conflict with each other, despite the race condition noted above.
2) After all active nodes in the round have selected some color, we iterate through the active nodes again. This time, each node checks whether it has any active neighbor with the same color. If a node v has an active neighbor w with the same color, we will recolor either v or w.
   To do this, we first compare v and w's degrees. If one node has a smaller degree, it loses its color and is added to next round's queue to be recolored. We recolor the lower degree node in order to emulate the largest-degree-first



(a) Initial random color generation (b) Resolving conflicts (c) Next random color generation (d) Next random color generation

Fig. 2. Illustration of CC. The value in each node is its value. **(a)** Assume we start from two initial colors. Each node is assigned to one of these two colors randomly. These assignments result in conflicts. **(b)** Resolve conflicts: A node loses its color if it has any neighbor that has the same color and higher degree. **(c)** Uncolored nodes are assigned random colors unused by neighbors. We still have a conflict after this step. **(d)** Previous steps are repeated until there are no conflicts.

node ordering, which often improves coloring quality. If v and w have the same degree, then the node with lower val loses its color and is enqueued for the next round. After all nodes have checked their neighbors for conflicts, we go back to step 1 and begin the next round, processing the nodes enqueued in the current round. If no nodes were enqueued, then all nodes have a color and the algorithm ends.

Since nodes always check for coloring conflicts after all of them have selected a color, the CC algorithm will run until no conflicts remain and will produce a valid coloring. The algorithm may not be work efficient, as a node may be enqueued several times. However, as we discuss in Section IV-B, in practice the amount of redundant work is low, and each node was processed an average of 1.3 times.

In CC, we update $max\text{-}color$ on the fly to estimate the graph's chromatic number. We considered other estimation methods, for example, running CJP on a small random subgraph and using its number of colors to bootstrap CC. However, we found these approaches gave inconsistent coloring quality, and in addition were slower than our current method.

C. *Implementation on the GPU*

We now discuss some issues related to efficiently implementing CJP and CC on the GPU.

Due to memory limitations, we perform coloring in *phases*, where in the k'th phase, $k \geq 0$, nodes are assigned colors in the range $256k + 1$ to $256(k + 1)$. In each phase, the main issue is how a node finds a color not used by its neighbors. This is done by first assigning a virtual warp of threads [31] to each node v to gather the colors used by v's neighbors. Each thread processes a disjoint subset of v's neighbors, and the thread maintains four 64-bit integer registers, which act as bitflags for the 256 colors in the current phase. If the thread sees that a color $c \in [256k+1, 256(k+1)]$ is used by one of v's neighbors, it sets the c'th bit among its 256 bitflags. After all threads have iterated through their portion of v's neighbors, the warp performs a reduction to OR together all the bitflags, thereby obtaining all the colors used by v's neighbors. Next, we use one thread from the warp to find the minimum unset bit (in CJP) or a random unset bit (in CC), using intrinsic bit operations in CUDA. It may be that all 256 bitflags for v are set, indicating that v's neighbors have used up all 256 permitted colors for the current phase. In this case, node v is placed in a queue to be colored in the next phase.

TABLE I
LIST OF GRAPHS USED IN EXPERIMENTS

Name	# Nodes	# Edges	Avg degree	Max degree
cant	63K	2M	64	78
economics	207K	0.7M	12	56
epidemiology	526K	1.1M	7	7
accelerator	121K	1.3M	22	81
QCD	49K	0.9M	42	42
harbor	47K	1.2M	51	145
scircuit	171K	0.5M	6	353
webbase-1M	1M	2.5M	5	28,687
spheres	83K	3M	72	81
ship	141K	3.9M	55	68
pwtk	218K	5.8M	53	180
protein	36K	2.2M	119	204
2cubes_sphere	101K	0.8M	16	31
cage12	130K	1M	16	33
hood	221K	5.4M	49	77
m133-b3	200K	0.4M	8	8
majorbasis	160K	0.9M	15	19
mario002	390K	1.1M	5	7
offshore	260K	2.1M	16	31
patents_main	241K	0.3M	5	212
poisson3Da	14K	0.2M	26	4,491
circuit5M	5.6M	29.8M	11	1,290,501
audikw_1	0.9M	38.8M	82	345
Bump_2911	2.9M	63.8M	44	195
rgg_n_2_24_s0	8.4M	265.1M	16	40
europe_osm	50.9M	54M	2	13
kron_g500-logn21	2.1M	91M	87	213,905
soc-LiveJournal1	4.8M	43.1M	18	20,334
hollywood-2009	1.1M	57M	100	11,468
indochina-2004	7.4M	152.2M	41	25,6425
nlpkkt200	16.2M	224.1M	28	28
nlpkkt240	28M	387.3M	28	28

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our algorithms. We ran the algorithms on 32 matrices from the University of Florida sparse matrix collection [15], listed in Table I. Since our coloring is on undirected graphs, we transformed any nonsymmetric matrix A to a symmetric one by computing $A + A^\top$. The graphs in the top part of the

TABLE II
AVERAGE RUNNING TIME (IN MILLISECONDS) AND NUMBER OF COLORS USED FOR DIFFERENT ALGORITHMS.

Graph	csrcolor		VEB		ColPack						CJP		CC	
	time	colors	time	colors	Largest-first		Smallest-last		Incidence-degree		time	colors	time	colors
					time	colors	time	colors	time	colors				
cant	10.9	112.0	17.1	36.1	9.1	27	11.6	23	11.4	25	4.0	31.0	3.5	40.6
economics	6.9	64.0	7.6	11.6	8.8	7	10.4	7	13.5	7	4.4	10.0	3.4	13.7
epidemiology	4.3	32.0	12.5	7.4	8.5	4	15.5	3	24.6	3	6.0	7.0	4.6	7.0
accelerator	5.7	79.0	9.7	22.5	9.9	20	13.5	18	14.7	18	5.7	20.0	3.4	23.5
QCD	5.1	80.0	7.7	15.0	4.8	2	6.0	2	5.6	2	3.9	8.0	2.1	17.5
harbor	10.5	139.0	19.6	55.1	7.8	53	8.1	49	7.9	48	9.1	51.0	2.6	58.0
scircuit	3.7	48.0	7.0	10.3	5.7	8	7.4	8	6.9	7	3.5	10.0	1.6	11.0
webbase-1M	110.2	208.0	34.4	63.1	21.0	62	21.2	63	22.0	63	98.8	62.0	17.2	67.0
spheres	15.7	128.0	24.7	46.1	11.8	30	16.7	33	18.3	33	8.5	42.0	4.6	49.5
ship	16.3	109.0	33.8	43.1	20.2	54	23.3	38	28.4	37	10.0	38.0	6.0	46.9
pwtk	22.0	96.0	51.5	39.2	25.8	42	28.3	35	32.8	40	10.0	37.0	9.1	43.5
protein	19.8	189.0	28.3	84.9	12.4	78	11.2	64	11.9	66	12.4	75.0	3.9	85.3
2cubes_sphere	3.5	48.0	5.9	13.5	9.3	12	12.4	10	12.4	10	3.5	12.0	2.2	14.3
cage12	4.0	64.0	7.0	15.4	8.7	13	13.7	12	14.1	13	4.8	14.0	2.7	16.2
hood	20.4	96.0	39.2	36.2	24.5	42	33.1	35	41.7	35	11.4	35.0	8.1	41.2
m133-b3	3.7	48.0	6.2	8.7	4.3	6	14.1	7	13.7	7	3.2	7.0	2.2	9.0
majorbasis	3.7	48.0	7.0	12.3	5.1	6	4.9	7	5.6	9	8.0	10.0	3.0	13.8
mario002	3.4	32.0	5.8	7.4	9.2	5	14.5	4	14.3	5	4.0	6.0	3.3	7.0
offshore	6.5	48.0	14.8	14.6	25.3	11	34.0	10	36.3	11	5.9	13.0	5.4	15.7
patents_main	5.6	64.0	5.7	15.5	12.2	10	14.3	9	14.8	9	6.3	13.0	2.2	14.6
poisson3Da	2.8	64.0	4.8	20.2	1.5	16	2.4	15	2.4	15	4.2	18.0	0.9	20.9
circuit5M	8,540.1	341.0	380.2	7.8	117.2	6	120.1	5	126.8	5	1.991.8	9.0	237.4	13.9
audikw_1	216.1	160.0	222.5	50.2	224.9	51	261.9	44	291.2	44	55.0	46.0	54.2	69.3
Bump_2911	203.8	96.0	332.1	32.2	350.3	31	449.2	25	518.1	28	94.2	29.0	115.3	37.0
rgg_n_2_24_s0	$\perp$	$\perp$	790.1	26.7	2,812.8	21	4,045.1	21	3,510.3	22	421.4	24.0	719.1	26.5
europe_osm	$\perp$	$\perp$	148.3	6.1	1,096.8	5	1,773.9	4	1,433.1	4	328.0	5.0	270.1	6.0
kron_g500-logn21	14,069.6	3,465.0	1,185.4	644.4	1,201.0	490	1,219.0	484	1,308.8	487	7,428.3	604.0	721.6	816.3
soc-LiveJournal1	610.8	557.0	279.8	332.6	1,012.7	323	1,069.8	321	1,218.7	325	299.0	331.0	217.0	339.7
hollywood-2009	1,779.5	2,317.0	665.8	2,209.0	499.1	2,209	453.5	2,209	490.6	2,209	2,930.5	2,209.0	252.5	2,209.0
indochina-2004	8,437.4	7,030.0	6,300.7	6,849.8	1,009.5	6,848	991.6	6,848	836.1	6,848	$\perp$	$\perp$	1,305.4	6,851.2
nlpkkt200	502.1	77.0	$\perp$	$\perp$	819.0	2	1,041.4	2	1,853.0	2	413.0	11.0	457.2	12.0
nlpkkt240	$\perp$	$\perp$	$\perp$	$\perp$	1,408.3	2	1,792.3	2	3,424.6	2	725.8	11.0	786.8	13.0

table are smaller, with up to 5.8M edges, while the graphs in the bottom part have up to 387M edges. Our GPU is an Nvidia Tesla K20Xm with 2688 cores in 14 SMs, peak single precision performance of 3935 GFlops/s and 250GB/s of memory bandwidth. Each SM has 48 KB of shared memory and 32K 64-bit registers. The CPU experiments were done on an 8-core Intel Xeon E5-2670 2.60GHz with 128GB of RAM running Linux CentOS 6.4.

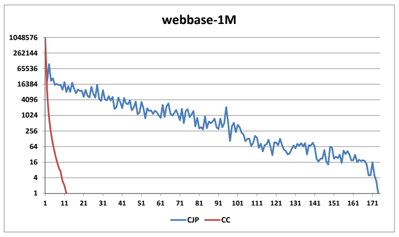
We compared our algorithms with csrcolor from Nvidia's cuSPARSE sparse matrix library [32], Deveci *et al*'s algorithms from [2], and Gebremedhin *et al*'s CPU-based algorithm in ColPack. Deveci *et al* proposed two algorithms called *vertex-based* and *edge-based*. For each graph, we show the running time from the faster of the two algorithms as well as the coloring quality of the faster algorithm. We label the result VEB, since either the vertex or edge based algorithm may be faster depending on the graph. The ColPack algorithm is single-threaded. It implements a number of node ordering heuristics, which we label *Largest first*, *Smallest last* and *Incidence degree* in our results. Since many of the algorithms are randomized, the reported running time and color usage for each graph are the average from 10 runs.

A. Throughput and number of colors

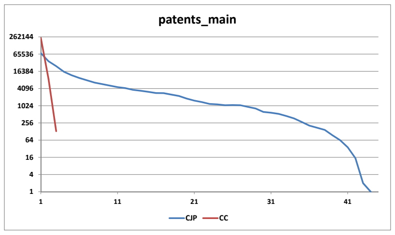
We first compare the running times of CJP and CC with the other algorithms. The results are shown in Table II. Note that on certain graphs some of the algorithms crashed. These are indicated by a $\perp$ in Table II.

CJP and CC are on average $1.5\times$ and $4.3\times$ faster than `csrcolor`, resp. Both CJP and CC showed the highest speedup on the graph *circuit5M*, where they were $4.3\times$ and $35\times$ faster, resp. Compared to VEB, CJP and CC were $2\times$ and $3.3\times$ faster on average, resp. CJP had the largest improvement on *pwtk*, where it was at $5.2\times$ faster, and CC performed best on *harbor*, where it was $7.6\times$ faster. Among the three heuristics we tested in ColPack, Largest-first was the fastest. CJP and CC were $1.9\times$ and $2.7\times$ faster, resp. than Largest-first on average, and up to $6.7\times$ and $5.5\times$ faster for some graphs.

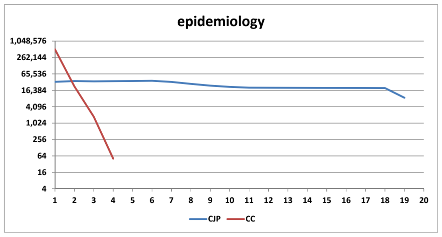
Next, we compare the algorithms in terms of the number of colors they used. The results are shown in Table II. In the two figures, the results have been normalized to the number of colors used by ColPack's Smallest-last ordering, since this usually led to the best coloring. `csrcolor` gave the worst results, often using many times more colors than Smallest-last. On average CJP used $4.3\times$ fewer colors than csrcolor and CC used $3.5\times$ less. CC used more colors than CJP for better coloring time; this trade-off can also be seen in ColPack framework where Largest-first uses $5 - 10\%$ more



(a) webbase-1M



(b) patents_main



(c) epidemiology

Fig. 3. Number of enqueued nodes (y-axis) in each round (x-axis) for *webbase-1M*, *patents_main* and *epidemiology*.

colors than Smallest-last, but is slightly faster. The number of colors used by VEB was intermediate between CJP and CC. On average VEB used 10% more colors than CJP and 10% fewer colors than CC. ColPack provided the best coloring quality, with the Smallest-last and Incidence-degree orderings typically the most effective. On average, CJP and CC used $1.5\times$ and $2\times$ more colors than Smallest-last, resp. However, these results are somewhat skewed by the presence of three bipartite graphs, *QCD*, *nlpkkt200* and *nlpkkt240*. Smallest-last colored these graphs optimally, whereas CJP and CC used $4-6.5\times$ more colors. If these outliers are removed, then CJP and CC used $1.3\times$ and $1.5\times$ more colors than Smallest-last, resp. Meanwhile, since bipartite graphs can be quickly colored using breadth-first search, we can add a heuristic to CJP and CC to quickly deal with this special case.

B. Work and efficiency

In this section, we analyze CJP and CC in terms of their parallelism and work efficiency. We also look at how quickly the value of $max\text{-}color$ converges in CC.

1) Parallelism and work: The amount of work in each round of CJP and CC depends on the number of nodes in the round's queue. Figure 3 shows the queue size in different rounds of CJP and CC on three graphs, *epidemiology*, *webbase-1M* and *patents_main*. These graphs were chosen because they exhibit large variations in the relative performance of CJP and CC.

CJP perform far more rounds than CC on all the graphs, finishing in 20, 175 and 45 rounds on *epidemiology*, *webbase-1M* and *patents_main*, resp., versus 5, 13 and 4 rounds for CC. Since each round is performed in a separate kernel, this created a large amount of overhead for CJP. CC was able to finish in a small number of rounds because its queue size decreased very quickly, dropping by a factor of 23, 108 and 27 after the first round on *epidemiology*, *webbase-1M* and *patents_main*, resp. We describe why this happened when we discuss the rate of increase in $max\text{-}color$ in the next section.

In addition to running for a large number of rounds, CJP also had relatively small queue sizes for a large portion of its execution in *webbase-1M* and *patents_main*. The queues had less than 4000 nodes for about 3/4 of the rounds in both cases. Such a small queue size may not provide enough work to fully occupy the GPU. As a result, CC performed substantially better than CJP on these graphs, being about $2.9\times$ and $5.7\times$ faster on *patents_main* and *webbase-1M*, resp. On the other hand, CJP was able to maintain relatively large queues of over 20K nodes in *epidemiology*. Here, CJP's performance was much closer to CC's, being only $1.3\times$ slower.

The CC algorithm may process a node multiple times due to coloring conflicts and thus perform redundant work. To measure the amount of redundancy on a graph, we summed up the sizes of the queues in all the rounds of CC's execution on the graph and divided this by the number of nodes in the graph. The average redundancy across all the test graphs was 1.3. This shows that conflicts are not very frequent, and that CC is almost as work efficient as CJP.

2) Rate of increase of $max\text{-}color$: Recall that in CC, nodes that cannot find an unused color increment $max\text{-}color$. In this section we look at how quickly $max\text{-}color$ increases on several graphs. A faster increase leads to fewer coloring conflicts and less redundant work. At the same time, we want to keep $max\text{-}color$ small enough to avoid wasting colors. Figure 4 shows the size of $max\text{-}color$ compared its final value at the end of the execution, as a function of the percent of nodes in the first round queue that have been colored, for the graphs *cant*, *harbor* and *webbase-1M*. These matrices were chosen because they show different patterns in $max\text{-}color$'s increase. Also, we focus on $max\text{-}color$'s change in the first round because it does not increase much in subsequent rounds.

In *cant*, $max\text{-}color$ increases quite quickly, and reaches 80% of its final value after processing about 1/3 of the nodes. For *harbor*, the growth is much slower, and it only reaches 80% after processing 3/4 of the nodes. The reason is, *harbor* being a small graph, we did not sort its nodes by degree, and so many low degree nodes occurred near the start of the execution. These nodes are able to color themselves using only a few colors, and so $max\text{-}color$ increased relatively slowly. In *webbase-1M*, we again did not sort the nodes by degree. The degrees followed a power law distribution, where most nodes have low degrees and a few nodes have much higher degrees. Thus as the nodes are processed we initially encounter mostly low degree nodes which can be colored using only a few colors. It is only after about half the nodes have been processed that we encounter some high degree nodes which require a large value of $max\text{-}color$, and so it is at this point that $max\text{-}color$ increased sharply to the its final value. After

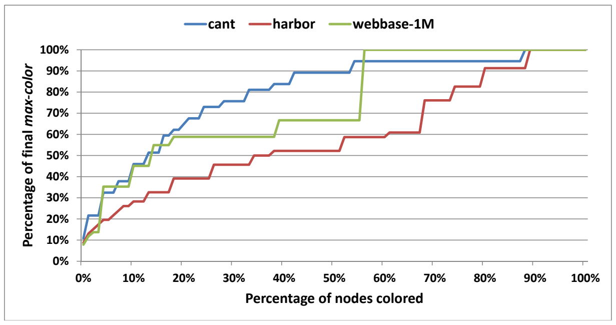


Fig. 4. Size of $max\text{-}color$ relative to its final value, as a function of the percent of nodes colored in the first round for *cant*, *harbor* and *webbase-1M*.

this large increase, all the nodes in the graph could more easily find unused colors, which improves CC's performance. On this graph CC was $6.4\times$ faster than csrcolor.

V. Conclusion

In this paper, we presented two algorithms for efficient graph coloring on the GPU. The CJP algorithm is a work efficient implementation of the Jones-Plassmann algorithm, and the CC algorithm optimistically colors nodes while adjusting the allowed color range dynamically. Both algorithms find high quality colorings while being substantially faster than state-of-the-art algorithms.

In the future, we hope to incorporate our coloring algorithms into GPU linear algebra packages to study the effects of coloring speed and quality on end-to-end linear algebra performance. We also want to further improve the performance of our algorithms through better load balancing, especially in scale-free networks with highly skewed degree distributions. Finally, we want to implement additional node ordering heuristics such as smallest-degree-last to improve coloring quality, and examine the tradeoff in coloring speed versus quality.

References

[1] 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, Nvidia Corp., Santa Clara, CA, Tech. Rep., 2015.

[2] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," *IEEE International Conference on Parallel and Distributed Processing (IPDPS)*, May 2016.

[3] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "Colpack: Software for graph coloring and related problems in scientific computing," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 1, p. 1, 2013.

[4] A. Gamst and W. Rave, "On frequency assignment in mobile automatic telephone systems," in *Proc. Globecom*, vol. 82, 1982, pp. 309–315.

[5] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther, "Efficient computation of sparse hessians using coloring and automatic differentiation," *INFORMS Journal on Computing*, vol. 21, no. 2, pp. 209–223, 2009.

[6] D. Hysom and A. Pothen, "A scalable parallel algorithm for incomplete factor preconditioning," *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2194–2215, 2001.

[7] M. Chiarandini, T. Stützle *et al.*, "An application of iterated local search to graph coloring problem," in *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, 2002, pp. 112–125.

[8] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19–37, 2015.

[9] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 147–156.

[10] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[11] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.

[12] Ü. V. Çatalyürek, 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.

[13] G. Rokos, G. Gorman, and P. H. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 414–425.

[14] J. Cohen and P. Castonguay, "Efficient graph matching and coloring on the gpu," in *GPU Technology Conference*, 2012, pp. 1–10.

[15] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[17] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[18] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[19] T. F. Coleman and J. J. Moré, "Estimation of sparse jacobian matrices and graph coloring blems," *SIAM journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209, 1983.

[20] D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[21] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," *SCCS-666*, pp. 1–19, 1995.

[22] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 2014, pp. 166–177.

[23] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[24] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the gpu and some techniques to improve load imbalance," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 610–617.

[25] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your jacobian? graph coloring for computing derivatives," *SIAM review*, vol. 47, no. 4, pp. 629–705, 2005.

[26] D. W. Matula, "A min-max theorem for graphs with application to graph coloring," in *Siam Review*, vol. 10, no. 4. Siam Publications 3600 Univ City Science Center, Philadelphia, PA 19104-2688, 1968, p. 481.

[27] G. Szekeres and H. S. Wilf, "An inequality for the chromatic number of a graph," *Journal of Combinatorial Theory*, vol. 4, no. 1, pp. 1–3, 1968.

[28] R. Diestel, "Graph theory. 2005," *Grad. Texts in Math*, vol. 101, 2005.

[29] T. Bjarne, "Colouring, stable sets and perfect graphs," *Handbook of combinatorics*, vol. 1, p. 233, 1995.

[30] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145832

[31] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 267–276. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941590

[32] "Nvida cusparse library," https://developer.nvidia.com/cusparse.