Improving the Speed and Quality of Parallel Graph Coloring

Ghadeer Alabandi

Texas State University, gaa54@txstate.edu

Martin Burtscher

Texas State University, burtscher@txstate.edu

Graph coloring assigns a color to each vertex of a graph such that no two adjacent vertices get the same color. It is a key building block in many applications. In practice, solutions that require fewer distinct colors and that can be computed faster are typically preferred. Various coloring heuristics exist that provide different quality versus speed tradeoffs. The highest-quality heuristics tend to be slow. To improve performance, several parallel implementations have been proposed. This paper describes two improvements of the widely used LDF heuristic. First, we present a "shortcutting" approach to increase the parallelism by non-speculatively breaking data dependencies. Second, we present "color reduction" techniques to boost the solution of LDF. On 18 graphs from various domains, the shortcutting approach yields 2.5 times more parallelism in the mean, and the color-reduction techniques improve the result quality by up to 20%. Our deterministic CUDA implementation running on a Titan V is 2.9 times faster in the mean and uses as few or fewer colors as the best GPU codes from the literature.

CCS CONCEPTS • Computing methodologies • Massively parallel algorithms

Additional Keywords and Phrases: Graph coloring, shortcutting, color reduction, parallelism, GPU computing

1 INTRODUCTION

Graph coloring refers to the assignment of colors (i.e., unique symbols) to the vertices of a graph such that no adjacent vertices have the same color. More formally, a vertex coloring of an undirected graph $G = (V, E)$ is a mapping C from vertices to colors such that $C(i) \neq C(j)$ for every edge $(i, j) \in E$. The graph coloring problem is the problem of coloring a graph using as few colors as possible without violating this adjacency constraint.

Graph coloring is an algorithmic building block in many applications such as clustering, data mining, image capturing, image segmentation, networking, resource allocation, process scheduling, optimizing the calculation of sparse Jacobian matrices [10], LU factorization [37], and parallel Gauss-Seidel algorithms for solving non-linear equations [28]. An example of resource allocation might be an ambulance service that uses graph coloring to schedule non-emergency transports as follows. First, they build a graph where each vertex represents a transport, and there is an edge between any pair of transports that overlap in time. Then, they color the graph. The result shows not only how many ambulances are needed (the number of unique colors) but also which transports should be handled by, say, the red ambulance (all transports that are colored red). The solution minimizes the number of required ambulances (the cost) and maximizes their utilization.

Graph coloring is NP-hard, i.e., there is no known polynomial time algorithm that can solve it optimally [20]. However, many heuristic algorithms exist to color a graph using few colors. These algorithms produce a valid coloring, meaning they guarantee that no adjacent vertices have the same color, but they may require more colors than the optimal algorithm, meaning they do not guarantee optimality. Moreover, these heuristics provide different tradeoffs between the coloring quality and the execution time. Typically, faster algorithms tend to require more colors. The problem we are addressing is how to deliver a very good coloring quality at high speed. Our solution is to increase the parallelism and post-processing the result to reduce the number of colors needed.

One well-known heuristic is the greedy algorithm. It assigns a random priority to each vertex. Then, it repeatedly selects the uncolored vertex that currently has the highest priority and colors it with the best available color, i.e., the first available color that is not already assigned to one of the vertex's neighbors. In graph coloring, the colors are typically ordered (first color, second color, etc.) and the first color is the "best" (most preferred) color.

Many parallel graph coloring algorithms [5] [9] [24] [41] follow the Jones-Plassmann approach [30], that is, they are based on the observation that any independent set of vertices can be colored in parallel. The strategy used for the coloring depends on the application. If fewer colors are desirable, the algorithm needs to emphasize the coloring quality at the cost of performance. If the application is runtime sensitive, the number of colors might be compromised in favor of a higher speed. Combining the Jones-Plassmann approach with different priority heuristics allows to select different points in this quality versus speed tradeoff space.

Several priority heuristics have been proposed for determining the order in which to color the vertices. Some of them can be implemented to run in linear time in the size of the graph. There are six prominent ordering heuristics for graph coloring: 1) first-fit ordering (FF), where the vertices are colored in the order in which they appear in the linearized input, 2) random ordering (R), where the vertices are colored in random order, 3) largest-degree-first ordering (LDF), where the vertices with larger degrees are colored first, 4) smallest-degree-last ordering (SDL), where the vertices with the smallest degree are successively removed from the graph, the modified graph is colored using the LDF heuristic, and finally the removed vertices are gradually re-inserted and colored, 5) saturation-degree ordering (SD), where the vertices whose colored neighbors have the largest number of unique colors are colored first (using the vertex degree as a tie breaker), and 6) incidence-degree ordering (ID), where the vertices with the largest number of colored neighbors are colored first irrespective of the number of unique colors (using the vertex degree as a tie breaker). Where needed, these heuristics include a tie breaker, which is often the vertex identifier. In general, LDF tends to produce better colorings than FF and R at the same performance level, SDL and SD tend to produce better colorings than LDF but with a large additional cost in runtime, and ID tends to produce similar coloring quality as LDF but is slower [26].

For any ordering heuristic, assigning the best available color to each vertex guarantees that the number of colors used is always bounded by d_{max}+1, where d_{max} is the highest degree of any vertex in the graph. However, some ordering heuristics, in particular SDL, lower this bound to a much smaller quantity called the degeneracy (or core) of the graph. Other ordering heuristics, such as LDF, provide a bound that lies somewhere in between.

Our algorithm is based on the Jones-Plassmann (JP) approach with the largest-degree-first (LDF) heuristic. We selected JP-LDF because it is widely used as it tends to produce good colorings while being quite fast [26]. It colors vertices with higher degrees first, so vertices with a lower degree must wait before the algorithm can assign a color to them. To reduce this waiting, we have developed "shortcuts" that, under certain conditions, allow us to non-speculatively color lower-degree vertices before their higher-degree neighbors have been colored [1]. Thus, the shortcuts increase the parallelism as more vertices can be colored simultaneously, which boosts performance. Importantly, the shortcuts are guaranteed to yield the same final coloring as the JP-LDF algorithm without the shortcuts.

The domains of some other NP-hard graph problems, notably the traveling salesman problem (TSP) [1], distinguish between two categories of heuristics: *construction* heuristics and *improvement* heuristics [27] [38]. Construction heuristics build a solution from scratch whereas improvement heuristics take a valid solution as input and try to make it better. The two "color-reduction" heuristics we propose in this paper are one of the first improvement

heuristics in the graph coloring domain. The LDF heuristic always assigns the best available color to a vertex. However, this may prevent the vertices that end up with the highest color h from getting a better color. Under some conditions, our color reduction heuristics can assign worse colors (below h) to other vertices to free up better colors for the vertices with color h. Recoloring them with one of the better colors lowers the number of required colors beyond the capabilities of LDF. Our color-reduction heuristics are guaranteed to never make the coloring quality worse. On about half of our inputs, they reduce the number of colors, thus boosting the solution quality. They are independent of LDF and can be used to improve the result of any construction heuristic.

This paper makes the following main contributions.

- It presents algorithmic shortcuts to increase the parallelism in graph coloring without affecting the coloring quality.
- It presents color-reduction heuristics to improve the solutions of other graph coloring heuristics.
- It describes techniques to efficiently implement and deterministically parallelize these shortcut and color-reduction algorithms.
- It demonstrates that our CUDA implementation is faster than prior CPU and GPU graph coloring codes on a variety of graphs while, on average and on most of the tested graphs, also using fewer colors.

The CUDA source code is available at https://cs.txstate.edu/~burtscher/research/ECL-GC/.

The rest of the paper is organized as follows. Section 2 provides background information on (parallel) graph coloring. Section 3 explains the shortcuts and optimizations to implement them efficiently. Section 4 explains the color-reduction heuristics and how they are parallelized. Section 5 summarizes related work. Section 6 describes the experimental methodology. Section 7 presents and analyzes the results. Section 8 concludes the paper.

2 BACKGROUND

Throughout this paper, we use the color order shown in Figure 1a, i.e., the first color (red) must be chosen whenever possible. If it is not available, the second color (blue) must be chosen if possible, and so on.

We use the graph in Figure 1b with 7 vertices and 16 edges for illustration. To simplify the discussion, the vertices are labeled in LDF order, meaning they are to be colored in alphabetic order. The resulting ordering imposes a direction upon each edge (from the higher-priority vertex that must be colored earlier to the lower-priority neighbor), which turns the undirected graph into the directed acyclic graph (DAG) shown in Figure 1c.

Figure 1d displays a valid coloring with four colors. This is the result that the greedy serial algorithm produces when processing the vertices alphabetically. It first colors A, which has no colored neighbors, so A gets red. Then B is colored, which is adjacent to A and, therefore, cannot be red. Hence, B is assigned blue, the next best color. Vertex C can be red again and D must take orange as it has red and blue neighbors. E must be purple as it has red, blue, and orange neighbors. Finally, F can be blue and G can be orange. Note that the serial algorithm requires as many steps as there are vertices. Each step must traverse all edges of the current vertex, resulting in the total work of $O(|V| + |E|)$ where $|V|$ is the number of vertices and $|E|$ the number of edges in the graph. Any parallel algorithm that adheres to the same vertex priority must produce the same coloring, including the JP algorithm and our algorithm, which we named "ECL-GC".

A DAG generally only specifies a partial order, in this case the order in which to color the vertices. The parallelism of the JP algorithm originates from this partial order. The depth of the DAG determines the number of parallel steps, and the width at a given level (when drawing the DAG top-down) determines the amount of parallelism.

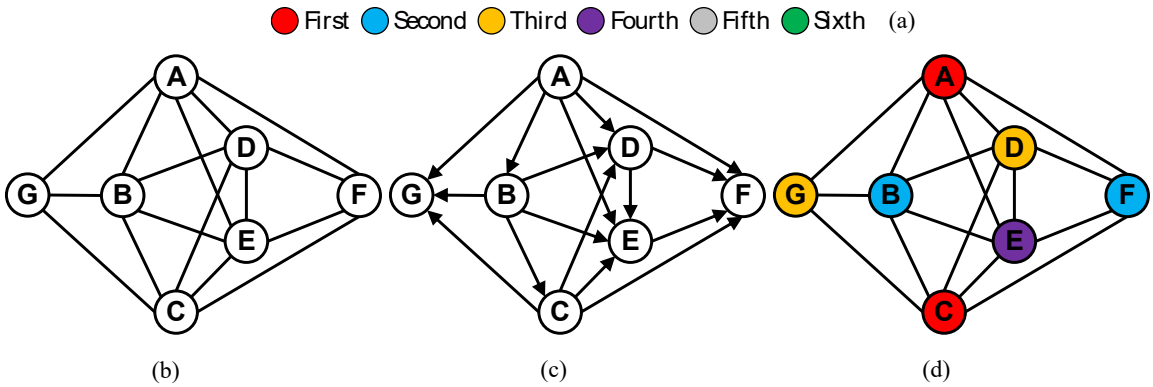


Figure 1: Assumed color priority order (a), sample graph (b), LDF-imposed DAG (c), and greedy coloring (d)

In many JP implementations, each vertex *v* starts out with a list of *k*+1 possible colors, where is *k* is the number of higher-priority neighbors, i.e., incoming DAG edges. This number suffices because, in the worst case, every incoming edge will be from a differently colored neighbor and use up the first *k* colors, leaving the *k*+1st color for vertex *v*. If the incoming edges end up not using all first *k* colors, because some neighbors of *v* either have the same color or use a color above *k*, then at least one of the first *k* colors will be available for *v*. Hence, it always suffices to only reserve the first *k*+1 colors for a vertex with *k* incoming edges [45]. Whenever a higher-priority neighbor is colored, that color is removed from the list. Each vertex is ultimately colored with the best remaining color.

JP often uses *bitmaps* for implementing these lists where each bit represents a different color [33]. A set bit "1" means the corresponding color is still possible and a cleared bit "0" means the color is no longer available. The position of the bit indicates to which color it refers. A colored vertex has a single set bit in the bitmap reflecting the color of the vertex. Uncolored vertices have at least two set bits. Whenever a higher-priority vertex is colored, the corresponding bit must be cleared in its lower-priority neighbors since that color is no longer available.

Figure 2 illustrates how the bitmap for vertex R changes as the neighbors with a higher priority get colored. The bitmap starts out with six possible colors since there are five higher-priority neighbors, indicated by the incoming arrows. To improve readability, we show each set bit in the color it represents.

Assuming vertex C is the first to be colored and it is colored blue, the blue bit in vertex R's list is zeroed out to indicate that blue is no longer a possible color for R. Next, assume vertex E is colored gray. Hence, the gray bit is set to zero in the bitmap. Then, vertex A is colored green, clearing the green bit. When vertex D is also colored green, the bitmap does not change because the green bit is already zero. Finally, vertex B is colored red and the red bit is cleared. Now that all higher-priority neighbors have been colored, we can color R. The remaining "1" bits in the bitmap indicate which colors can be used (yellow and purple in the example). Since JP-LDF always uses the best available color, vertex R is colored yellow. This can be achieved, for example, by zeroing out all bits above the first "1" in the bitmap as shown in Figure 2.

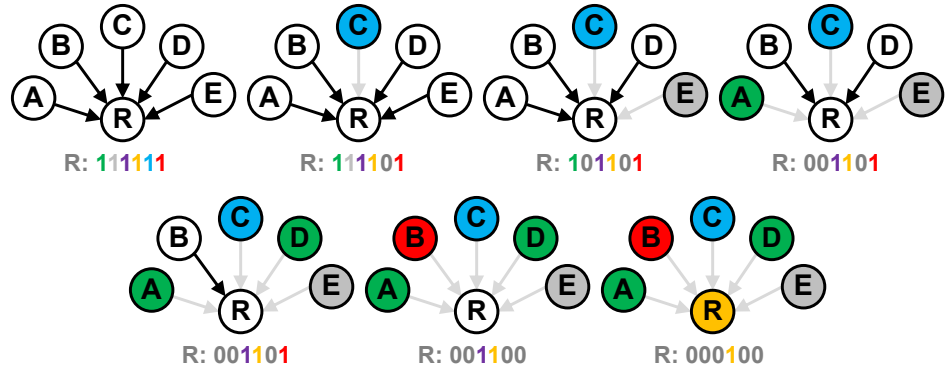
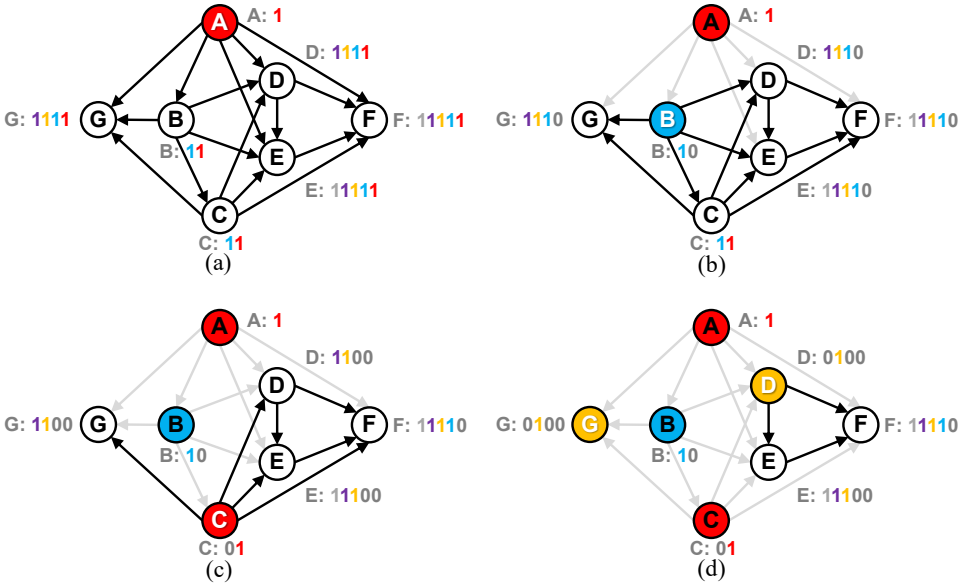


Figure 2: Bitmap of vertex R representing the list of remaining possible colors

Figure 3 illustrates the steps of the JP-LDF algorithm on our sample graph. Figure 3a shows the initialization step, which computes the direction of each edge in parallel by comparing the degrees of the two vertices the edge connects and invoking the tie breaker if needed. Vertex A can already be colored as it has no incoming edges. In each of the following processing steps, every uncolored vertex checks, in parallel, whether all its higher-priority neighbors (incoming edges) have been colored. We visualize this with light edges. Once a vertex has no incoming dark edges, it can be colored.



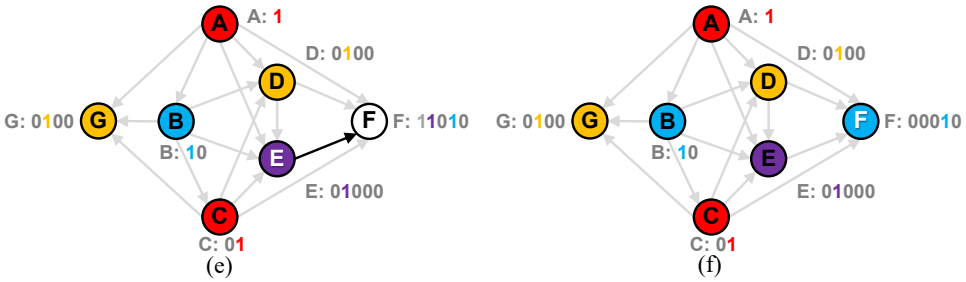


Figure 3: Initialization and computation steps of the parallel JP-LDF graph-coloring algorithm

In the first processing step (Figure 3b), all neighbors of vertex A see that A has been colored and zero out the red bit in their bitmaps. At this point, vertex B has no incoming dark edges anymore. It gets blue as that is the best available color in its bitmap. In the second step (Figure 2c), all lower-priority neighbors of vertex B clear their blue bits. This colors vertex C red. In the third step (Figure 2d), the lower-priority neighbors of vertex C see that C has been colored red, but the bitmaps do not change as none of them have the red bit set. Moreover, vertices D and G find that all their higher-priority neighbors have been colored. So, they are colored concurrently with the best available color, which happens to be orange in both cases. In the fourth step (Figure 2e), all lower-priority neighbors of vertices D and G zero out their orange bit. This colors vertex E purple. In the fifth and final step (Figure 2f), the purple bit is cleared in vertex F's bitmap and F is colored with the best remaining color, which is blue. Since all vertices are now colored, the JP-LDF algorithm terminates. It takes JP-LDF five steps to complete because the longest dependence chain in the DAG has five edges (A→B→C→D→E→F).

3 SHORTCUT APPROACH

There is little parallelism in the JP example from the previous section. Only one step colors more than one vertex. Yet, additional *non-speculative* parallelism may exist. To see where it resides, consider the partially colored subgraph in Figure 4a. We reuse the color priorities from Figure 1a in this section.

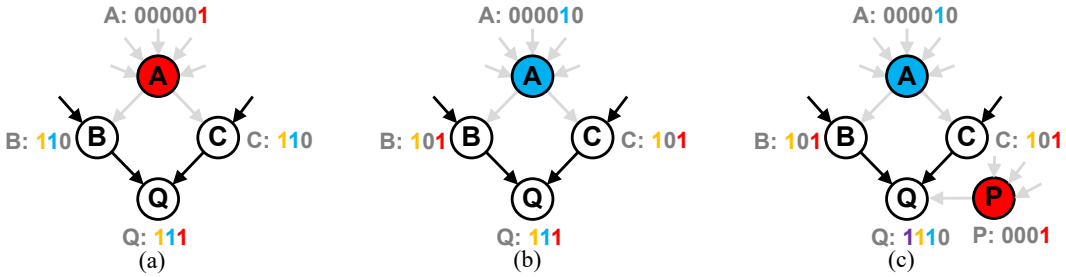


Figure 4: Examples of Shortcut 1

Vertices B and C cannot be colored because they both have a higher-priority neighbor that has not yet been colored, as indicated by the incoming dark edge. It appears that vertex Q also cannot be colored for the same reason. However, it can safely be colored red (the best color) without waiting for B or C. This is safe because B and C are guaranteed not to use red since they both have a neighbor that is already red. Figure 4b depicts a similar scenario but vertex A is now blue. Applying the same reasoning, we conclude that it is safe to color vertex Q blue as well. But

we must give each vertex the best possible color, i.e., the same color as the serial and JP-LDF algorithms. Unfortunately, we do not yet know whether it is possible to color vertex Q red and must, therefore, wait. In the modified case depicted in Figure 4c, we do not have to wait because blue is now the best possible color for Q, and we know that neither B nor C will be blue. Generalizing these observations leads to the first enhancement we propose, which we call a "shortcut" because it allows the coloring of vertices before it is their turn.

Shortcut 1: A vertex can safely be colored with its best possible color as soon as its uncolored higher-priority neighbors are no longer considering that color.

To be able to determine whether this is the case, we need to know what colors each vertex is still considering. Luckily, this information is stored in the bitmaps, which are already present in many JP-based graph-coloring codes to find the best available color when it is time to color a vertex [33]. Our approach uses these bitmaps for two additional tasks, namely, to determine the best available color of a vertex (the lowest set bit in its bitmap) before it is time to color the vertex and to determine if any of the higher-priority neighbors are still considering that color. Together, these two pieces of information allow us to decide whether the first shortcut can be applied.

We also use the information in the bitmaps for a second type of shortcut. The second shortcut makes it possible to ignore some higher-priority neighbors before they have been colored, which is tantamount to deleting an edge from the graph. This has two benefits. First, it enables the removal of one possible color from the bitmap since the number of incoming edges has decreased by one, which may make the first shortcut more effective (on other vertices). Second, it speeds up later processing steps as they no longer need to check the deleted edge. Figure 5 illustrates the idea behind the second shortcut.

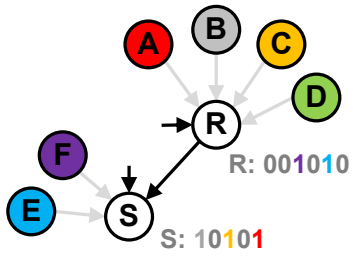


Figure 5: Example of Shortcut 2

In this example, vertex R cannot be colored because it is waiting for one higher-priority neighbor (the incoming dark edge). However, we already know it will end up with either blue or purple as those are the only two possible colors remaining. Similarly, vertex S cannot be colored yet, and we know that its remaining possible colors are red, orange, and gray. Since there is no overlap between the possible colors of vertices R and S, no matter which of its possible colors R eventually obtains, it will not interfere with S. Hence, we can safely delete the edge from R to S. Doing so lowers the number of incoming dark edges of vertex S to one, meaning it only needs to consider two possible colors. Consequently, we can remove the worst color from its list of possible colors, which is gray. Generalizing this idea leads to the second shortcut.

Shortcut 2: An edge from a higher-priority vertex u to vertex v can safely be removed as soon as there is no overlap between the possible colors of vertices u and v, which enables the removal of the worst color from the list of possible colors of vertex v.

Correctness proof of the two shortcuts:

1. Our approach augments the JP algorithm with two shortcuts. Therefore, it produces the same coloring as the JP algorithm when disabling the shortcuts.
2. Neither shortcut applies to the initialization phase, which is why the bitmaps start out identically with and without the shortcuts.
3. During processing, the only operation performed on the bitmaps, with and without the shortcuts, is the clearing of bits. This *monotonicity property* guarantees that, once there is no overlap between the first set bit (Shortcut 1) or any set bit (Shortcut 2) and another bitmap, there never will be an overlap anymore.
4. Only overlapping set bits can result in the clearing of the first set bit in a lower-priority vertex's bitmap (namely when the higher-priority vertex is colored with that color). Consequently, the first lower-priority set bit will never be cleared once the corresponding bit in all higher-priority vertices is zero (Shortcut 1).
5. Once there is no overlap, a higher-priority vertex cannot possibly select a color that a lower-priority vertex considers. Hence, the connecting edge can be ignored (Part 1 of Shortcut 2). Since the higher-priority vertex will necessarily either select a color above k+1 or a color that another neighbor of the lower-priority vertex has already chosen (since there is no overlap in the bitmaps), the worst-case scenario cannot occur anymore, which is why we can eliminate the highest set bit of the lower-priority vertex (Part 2 of Shortcut 2).

It is important to note that neither of the two shortcuts affects the ultimate coloring of the graph. They just speed up the processing by increasing the parallelism. Figure 6 illustrates this on our sample graph.

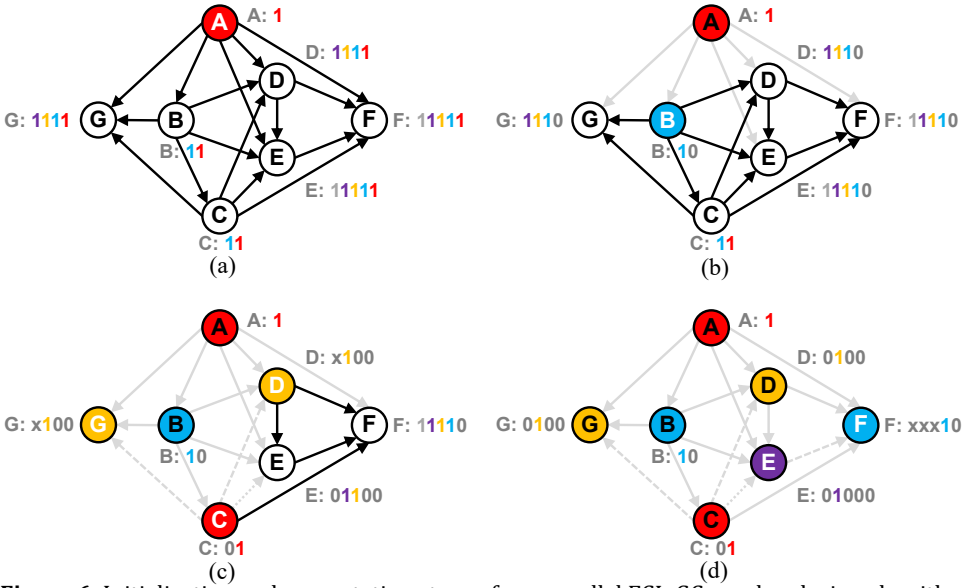


Figure 6: Initialization and computation steps of our parallel ECL-GC graph-coloring algorithm

The initialization phase (Figure 6a) of ECL-GC, our shortcut-based graph coloring algorithm, is identical to that of the JP-LDF algorithm (cf. Figure 3a). In particular, the bitmaps are initialized in the same way, and vertex A is already colored red. In each of the following computation steps, all uncolored vertices can be processed in parallel. Every vertex *v* visits its higher-priority neighbors. There are three cases to consider.

1) If a neighbor *u* has been colored, i.e., its bitmap only contains a single set bit, the edge from *u* to *v* is removed from consideration (grayed out) and one bit in the bitmap of *v* is cleared. If the bit corresponding to *u*'s color is set, that bit must be cleared since this color is no longer a possible color for *v*. This is equivalent to the standard JP algorithm. However, if the bit corresponding to *u*'s color is not set, the highest set bit in the bitmap of *v* is cleared instead. This is not done in the JP algorithm. It is optional in ECL-GC but may help with the following two cases.

2) If a neighbor *u* has not yet been colored, i.e., its bitmap contains multiple set bits, and none of the set bits in *u*'s bitmap overlap with the set bits in v's bitmap, the edge from *u* to *v* is removed from consideration (grayed out) and the highest set bit in the bitmap of *v* is cleared. This implements Shortcut 2.

3) For all uncolored higher-priority neighbors, the union (bitwise OR) of their bitmaps is computed. If the currently best possible color of *v* is not in the union, all incoming edges are removed from consideration and *v* is colored with its best available color, i.e., all bits above the lowest set bit are cleared (since that many edges were removed). This implements Shortcut 1.

In the first computation step of ECL-GC (Figure 6b), all vertices that are adjacent to A clear their red bit. Note that this colors vertex B as it only has one set bit left. B gets blue because the set bit is in the second position.

In the next computation step (Figure 6c), multiple events occur. All uncolored vertices that are adjacent to B clear their blue bit. This colors vertex C red. Due to the parallel processing, the other vertices either see the old bitmap of "11" or the new bitmap of "01" for C. Either view suffices for vertices D and G, both of which have vertex C as the only remaining higher-priority neighbor, to conclude that they can be colored orange using Shortcut 1 since their best possible color is not considered by any of their higher-priority neighbors. Applying Shortcut 1 clears all bits past the first set bit, indicated by an "x" in the figure. Shortcut 2 can also be applied in this computation step. The bitmap of E has no overlap with the (old or new) bitmap of C, so the edge from C to E is removed from consideration, and the highest set bit of E is cleared.

In the third computation step (Figure 6d), vertices E and F remove their orange bit due to vertex D, which colors vertex E purple. Vertex F may not yet see this update of vertex E's bitmap but can still conclude that its first set bit is not contained in any of its remaining neighbors' (vertices C and E) bitmaps, that is, it can be colored blue using Shortcut 1 and the higher bits are cleared. At this point, all vertices are colored, so the algorithm terminates.

The resulting coloring is identical to that of the serial and JP-LDF algorithms. Moreover, due to the increased parallelism, it only takes the ECL-GC algorithm three computation steps to color this graph compared to five steps for the JP-LDF algorithm.

3.1 Shortcut Derivation

The two shortcuts were systematically derived from combinations of intersections between the possible colors among neighboring vertices. Assume set $C(v) \subset \mathbb{N}$ contains the possible colors of vertex *v*. As shortcuts only apply to uncolored vertices and a vertex can only have a finite number of incoming DAG edges, $2 \leq |C(v)| < \infty$ holds. Thus, the complement $C'(v) = \mathbb{N} \setminus C(v)$ must have cardinality $|C'(v)| = \infty$. If $U(v)$ denotes the union of the possible colors of all uncolored higher-priority neighbors of *v*, $2 \leq |U(v)| < \infty$ must also hold and there must be at least one higher-

priority neighbor given that v is uncolored. Assuming vertex n represents one of those neighbors and that $B(v) \subset C(v)$ denotes the best color of $C(v)$, i.e., $|B(v)| = 1$, we end up with the 16 possibilities listed in Table 1.

Some intersections cannot yield an empty set due to the cardinality constraints outlined above. Others may yield an empty set, but the condition under which they do is not strong enough to yield a useful shortcut. The remaining four (red) cases are candidates. The 1st case from the top is Shortcut 2. The 5th case by itself is insufficient and only part of Shortcut 1. The 9th case is unnecessarily strong and already covered by the 13th case, which is Shortcut 1. We similarly tried using the possible colors of the neighbors' neighbors but could not find any exploitable shortcuts.

Table 1. Bitmap intersections and resulting actions

intersection	meaning of empty intersection	resulting action
$C(v) \cap C(n)$	poss. colors don't overlap with neighbor	remove edge (Shortcut 2)
$C'(v) \cap C(n)$	there is overlap: $C(n) \subset C(v)$	continue
$C(v) \cap C'(n)$	there is overlap: $C(v) \subset C(n)$	continue
$C'(v) \cap C'(n)$	impossible	
$B(v) \cap C(n)$	best color not considered by neighbor	record info (for Shortcut 1)
$B'(v) \cap C(n)$	impossible	
$B(v) \cap C'(n)$	best color is considered by neighbor	continue
$B'(v) \cap C'(n)$	impossible	
$C(v) \cap U(v)$	p. colors don't overlap with any neighbor	use best color (Shortcut 1)
$C'(v) \cap U(v)$	there is overlap: $U(v) \subset C(v)$	continue
$C(v) \cap U'(v)$	there is overlap: $C(v) \subset U(v)$	continue
$C'(v) \cap U'(v)$	impossible	
$B(v) \cap U(v)$	best color not considered by any neighbor	use best color (Shortcut 1)
$B'(v) \cap U(v)$	impossible	
$B(v) \cap U'(v)$	best color is considered by some neighbor	continue
$B'(v) \cap U'(v)$	impossible	

3.2 ECL-GC Implementation & Optimization

A direct implementation of the ECL-GC algorithm as described above may be inefficient due to long bitmaps that must be processed for vertices with many higher-degree neighbors. This potential inefficiency is concerning since the goal of the shortcuts is to accelerate the computation.

Graph coloring is typically performed on sparse graphs (e.g., dependence graphs) as there is little to be gained from coloring dense graphs that require close to $|V|$unique colors. We define a graph as sparse if it has $O(|V|)$ edges, that is, $|E| = c|V|$ where c is a constant (the average degree) that is much smaller than $|V|$. In a sparse graph, most of the vertices must have a low degree (much lower than $|V|$). Since a vertex of degree k can always be colored with one of the first k+1 colors, most vertices in sparse graphs can, therefore, be colored with just a few colors (typically significantly fewer than c). This observation led us to a design that treats high-degree and low-degree vertices separately. Specifically, we fully implement the presented shortcuts on the low-degree vertices but only approximate them on the high-degree vertices to avoid the costly processing of long bitmaps.

For each low-degree vertex with a degree under 32, we use a fixed bitmap with 32 bits (i.e., an integer). For all other vertices, we maintain the full bitmap to ultimately assign the best possible color as the conventional JP algorithm does but only use two integers for the shortcut computations. The first integer specifies the best possible color and the second integer the worst possible color. We do not update the worst possible color as we found that,

for high-degree vertices, it rarely gets small enough to matter before the vertex is colored. However, we maintain the best possible color precisely.

The shortcuts are approximated as follows with the two integers. Shortcut 1 is applied if the best possible color of a lower-priority vertex is not in the range between the best and worst possible color of any of the uncolored higher-priority neighbors. This simplification makes the Shortcut 1 processing independent of how long the bitmaps are but may miss some shortcutting opportunities. Shortcut 2 can be applied if the range between the lowest and highest possible color of a lower-priority vertex does not overlap with the range between the lowest and highest possible color of an uncolored higher-priority neighbor. We ended up not including the second shortcut for high-degree vertices in our implementation as our tests showed that it applies too infrequently to yield a speedup.

Our ECL-GC CUDA implementation consists of three kernels. Their operations are presented in Algorithms 1, 2, and 3 and explained in the following paragraphs.

The initialization kernel, outlined in Algorithm 1, sets the color of each vertex to zero (line 3) and puts all vertices of degree 32 or higher on a worklist (line 5). Furthermore, it records the incoming edges from higher-priority vertices (i.e., it builds the DAG, line 10) and initializes the bitmaps with the first k+1 colors (line 13). For efficiency reasons, this is done at thread granularity for the low-degree vertices and at warp granularity for the high-degree vertices. We use the unique vertex IDs to break ties when computing the priorities. This makes the code *deterministic*, meaning it always computes the same solution for a given input graph, independent of the GPU it is executed on and independent of internal thread and warp timing. The two for-each loops are parallelized. The worklist is updated concurrently using atomic instructions. The DAGs are generated in parallel using warp-based reductions.

ALGORITHM 1: ECL-GC Initialization

```
1:  worklist ← Ø
2:  for each vertex v in G do
3:      color_v ← 0
4:      if degree(v) ≥ 32 then
5:          worklist ← worklist ∪ v
6:      else
7:          DAGin_v ← Ø
8:          for each neighbor n of v in G do
9:              if priority(v) < priority(n) then
10:                 DAGin_v ← DAGin_v ∪ n
11:             end if
12:         end for
13:         posscol_v ← {0, 1, …, |DAGin_v|}
14:     end if
15: end for
```

The second kernel, outlined in Algorithm 2, processes the high-degree vertices, which it retrieves from the worklist (line 3). It can safely ignore the low-degree vertices as they are guaranteed to have lower priorities. This kernel uses persistent threads [25], meaning it is launched with only as many threads as can simultaneously be active on

the GPU rather than with as many threads as there are vertices in the graph. Each thread repeatedly and *asynchronously* loops over the vertices assigned to it and performs the computation steps until all vertices have been colored. To improve performance, the threads that have a vertex in need of processing (line 4) enlist the remaining threads in the warp to help traverse that vertex's incoming DAG edges (line 7). Hence, this kernel combines thread-based and warp-based parallelization. For each colored neighbor, the color is removed from the list of possible colors (line 12). If either the best color is removed (line 9) or the range of possible colors of the higher-priority neighbor overlaps with the best color (line 15), the shortcut is disabled. If all neighbors have been colored or the shortcut can be applied, the vertex is colored with its best possible color (line 22). Otherwise, the code will have to try to color this vertex again later (line 24). The two for-each loops are parallelized. The bitmaps (posscol) are updated using atomic instructions. The flags are updated using warp-based reductions.

ALGORITHM 2: ECL-GC High-degree Vertex Coloring

```
1:  do
2:      again ← false
3:      for each vertex v in worklist do
4:          if !colored(v) then
5:              shortcut ← true
6:              done ← true
7:              for each neighbor n of v in DAGin_v do
8:                  if colored(n) then
9:                      if color_n = bestcol_v then
10:                         shortcut ← false
11:                     end if
12:                     posscol_v ← posscol_v \ color_n
13:                 else
14:                     done ← false
15:                     if bestcol_n ≤ bestcol_v and bestcol_v ≤ worstcol_n then
16:                         shortcut ← false
17:                     end if
18:                 end if
19:             end for
20:             bestcol_v ← best(posscol_v)
21:             if done or shortcut then
22:                 color_v ← bestcol_v
23:             else
24:                 again ← true
25:             end if
26:         end if
27:     end for
28: while (again)
```

The third kernel, outlined in Algorithm 3, processes all vertices (line 3), skipping the ones that have already been colored (line 4), which includes all high-degree vertices. Hence, it suffices to only use 32-bit bitmaps in this kernel to fully implement both shortcuts. Since we know that the remaining uncolored vertices only have few higher-priority neighbors, this kernel performs all work exclusively at the thread level. As in the previous kernel, each thread asynchronously loops over the vertices assigned to it and performs the computation steps until its vertices have been colored. For each vertex, the thread traverses the incoming DAG edges (line 6) and computes the union of the possible colors of all uncolored higher-priority neighbors (line 7). If applicable (line 8), it applies Shortcut 2 (lines 9 and 10). If the neighbor has been colored (line 11), it removes the neighbor from further consideration (line 12) and removes the neighbor's color from the list of possible colors (line 13). If there are no uncolored higher-priority neighbors left or Shortcut 1 can be applied (line 16), the vertex is colored with its best possible color (line 17). Otherwise, the code will have to try to color this vertex again later (line 19). Only the outer for-each loop is parallelized as the inner for-each loop is guaranteed to only perform few iterations. This kernel's only synchronization is via producer-consumer relationships using volatile variables.

ALGORITHM 3: ECL-GC Low-degree Vertex Coloring

```
1:  do
2:      again ← false
3:      for each vertex v in G do
4:          if !colored(v) then
5:              union ← Ø
6:              for each neighbor n of v in DAGin_v do
7:                  union ← union ∪ posscol_n
8:                  if posscol_v ∩ posscol_n = Ø then
9:                      DAGin_v ← DAGin_v \ n
10:                     posscol_v ← posscol_v \ worst(posscol_v)
11:                 else if colored(n) then
12:                     DAGin_v ← DAGin_v \ n
13:                     posscol_v ← posscol_v \ color_n
14:                 end if
15:             end for
16:             if DAGin_v = Ø or union ∩ best(posscol_v) = Ø then
17:                 color_v ← best(posscol_v)
18:             else
19:                 again ← true
20:             end if
21:         end if
22:     end for
23: while (again)
```

For performance reasons, all set operations are implemented with logical bit instructions (AND, OR, and NOT). Moreover, the colors are allocated from MSB to LSB in the bitmaps, which may seem counterintuitive. This ordering does not affect the set operations, but it does accelerate the two important remaining operations: finding the first set bit (determining the best possible color, lines 16 and 17) and clearing the last set bit (removing the worst color, line 10). The position of the first set bit can quickly be obtained with the "count leading zero bits" instruction, which is available on many architectures, including GPUs. In contrast, a "count trailing zero bits" instruction is often not present. Clearing the last set bit of a value x can be done quickly by computing x &= (x – 1), which works irrespective of where the last set bit is located [44]. Generally, no equally fast way of clearing the first set bit of a value exists.

Our CUDA implementation has fewer than 300 statements with around 150 kernel statements and is available at https://cs.txstate.edu/~burtscher/research/ECL-GC/. It incorporates the above optimizations. It transfers the graph to the GPU and the computed colors back to the CPU. After initialization, the code repeatedly processes the vertices until convergence is reached. As mentioned, the processing is done asynchronously, which may result in threads reading outdated bitmaps. However, the bitmaps only ever have bits cleared. Similarly, for the larger-degree vertices that use the two integers bestcol and worstcol (in Algorithm 2) to implement the shortcut, bestcol only ever increases and worstcol stays constant. Due to these monotonicity conditions, it is always safe for a thread to act upon an outdated bitmap or bestcol value, but doing so may lead to extra iterations.

4 COLOR-REDUCTION HEURISTICS

As discussed in Section 1, LDF tends to be fast and yields a good coloring quality, but its quality is not as high as that of some slower ordering heuristics. To boost the coloring quality beyond the abilities of LDF without resorting to overly slow processing, we designed two *improvement* heuristics that take the solution of a graph-coloring algorithm such as JP-LDF as input and try to enhance it by reducing the number of colors used.

The high-level idea behind our color-reduction heuristics is the following. For each vertex v with the highest color, try to recolor some of its neighbors to free up a lower color and assign the freed-up color to v. If the heuristic succeeds in doing this for all vertices with the highest color, the new solution will use one fewer color. The procedure can be repeated until no further reduction is possible.

We illustrate how this works on the graph in Figure 7. For simplicity, we labeled the vertices according to their LDF priorities, meaning vertex A is colored first, then vertex B, etc. Reusing the color priorities from Figure 1a, we find that LDF colors this graph with three colors as shown in Figure 7a. Vertices D and E both end up with the highest color yellow. We can only reduce the total number of colors if we manage to recolor both vertices with a lower color. Note that they are yellow because they have higher-priority neighbors that already use all lower colors. Hence, we must first recolor at least one of their neighbors to free up a lower color. Figure 7b shows how this can be achieved by coloring vertex C blue, which frees up red to be used by vertices D and E. The recolored solution requires only two colors instead of three. It is important to note that LDF never even considered blue as a possible color for vertex C because vertex C has only one bit in its bitmap. In contrast, our color-reduction heuristics consider all colors that are better than the highest color for all neighbors of the highest-color vertices. This allows them to improve the solution in some cases.

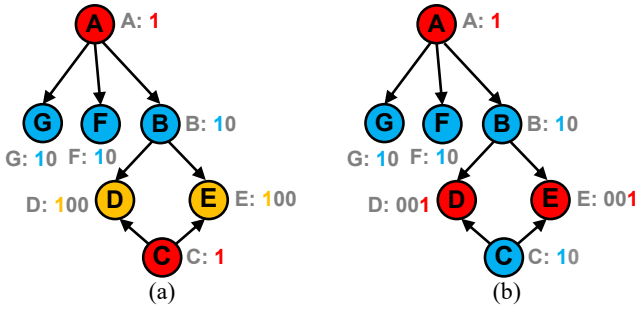


Figure 7: High-level idea behind color reduction

When trying to, e.g., recolor vertex D, we iterate over all its neighbors and choose one that can be recolored with the lowest color. The new color for the neighbor will necessarily be higher than its current color (since all lower colors are unavailable) but must be lower than the color of D to ultimately result in a savings. In Figure 7a, neighbor B cannot be recolored since its range of potential alternate colors (higher than blue but lower than yellow) is empty. However, neighbor C can be recolored. Its range consists of the color blue. As we already saw, using blue for vertex C frees up red for vertices D and E. Since we managed to recolor all highest-color vertices with a lower color, the new solution requires one fewer colors. Repeating this procedure does not yield any further improvement.

4.1 Two Graph-coloring Improvement Heuristics

This subsection describes our two color-reduction heuristics in detail. One of them targets low-degree graphs, such as road networks, and the other targets graphs with some high-degree vertices, such as power-law graphs. Both can be used in combination. We implemented the two heuristics in CUDA and added them as a post-processing stage to our graph-coloring algorithm ECL-GC. In the following text, the abbreviation "hic" stands for "highest color".

4.1.1 Heuristic 1

Heuristic 1 is intended for graphs with a few high-degree vertices. It checks whether all neighbors of the hic vertices that currently use color x can be recolored to y, where $y < \text{hic}$. If such a pair of x and y values exists, performing the corresponding recoloring will free up color x to be used by all hic vertices.

Since Heuristic 1 recolors all hic vertices to the same new color, it tends to only succeed on graphs with relatively few hic vertices. Low-degree graphs typically have a small range of used colors and many hic vertices, making it unlikely that we can find a single replacement color that works for all hic vertices. Hence, we only recommend Heuristic 1 for graphs with high-degree vertices.

Figure 8 provides an example of how Heuristic 1 works. Figure 8a shows an LDF colored graph where vertex G has the highest color, making it the only hic vertex. By examining G's neighbors, we find that none of the blue and yellow neighbors can be recolored to any color less than hic. However, all red neighbors (there is only one) can be recolored to yellow, which allows the hic vertex G to be colored red, as shown in Figure 8b. Thus, Heuristic 1 is able to reduce the number of colors from four to three in this example.

For performance and parallelization reasons, we implemented Heuristic 1 as presented in Algorithm 4. It first populates a worklist with all vertices that have a hic neighbor (lines 2 through 9). Next, it creates a 2-dimensional Boolean matrix of hic-by-hic size and initializes all elements to true (line 10). Then, it sets the matrix elements to

false that correspond to a pair <*x*, *y*> of colors such that *x* is the color of a vertex *v* from the worklist and *y* is the color of a neighbor of *v* (lines 11 through 15). If at least one matrix element remains true (line 19), its coordinates are recorded in the pair <*x*, *y*> (line 20). If multiple true elements remain, we deterministically pick the one with the lowest coordinates (not shown). Assuming at least one true element exists (line 24), all vertices from the worklist whose color is *x* are recolored to *y* (lines 25 through 29) and all hic vertices are recolored to *x* (lines 30 through 34). All for-each loops of Algorithm 4 can be executed in parallel. The worklist is populated using atomicAdd instructions. Setting matrix elements to false does not require synchronization. Deterministically finding the lowest true matrix element can be done with atomicMin instructions. Recoloring the vertices is embarrassingly parallel.

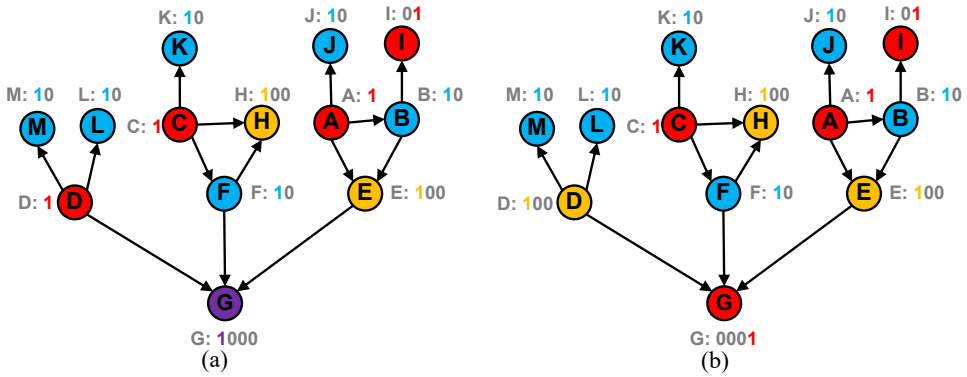


Figure 8: Example of Heuristic 1

ALGORITHM 4: Color-reduction Heuristic 1

```
1:  hic ← highest used color
2:  worklist ← Ø
3:  for each vertex v in G do
4:      for each neighbor n of v in G do
5:          if colorn = hic then
6:              worklist ← worklist ∪ v
7:          end if
8:      end for
9:  end for
10: m ← Boolean hic × hic matrix initialized to true
11: for each vertex v in worklist do
12:     for each neighbor n of v in G do
13:         m[colorv, colorn] ← false
14:     end for
15: end for
```

```
16: x, y ← hic, hic
17: for each i form 0 to hic - 1 do
18:     for each j form 0 to hic - 1 do
19:         if m[i, j] then
20:             x, y ← i, j
21:         end if
22:     end for
23: end for
24: if x, y ≠ hic, hic then
25:     for each vertex v in worklist do
26:         if color_v = x then
27:             color_v ← y
28:         end if
29:     end for
30:     for each vertex v in G do
31:         if color_v = hic then
32:             color_v ← x
33:         end if
34:     end for
35: end if
```

4.1.2 Heuristic 2

Heuristic 2 is intended for low-degree graphs. It checks, for each hic vertex *v*, whether its neighbors that currently use color *x* can be recolored to *y*, where *y* < hic. If such a pair of *x* and *y* values exists that does not conflict with any other such pair (see below), performing the corresponding recoloring will free up color *x* to be used by vertex *v*.

Unlike our first heuristic, this heuristic allows the hic vertices to be recolored with different colors, making it more likely to succeed on graphs with many hic vertices. However, it requires more memory. To cap the memory usage, we only apply Heuristic 2 to the first 32 colors, which typically suffices on low-degree graphs.

Recoloring graphs using multiple new colors may result in two kinds of conflicts that must be prevented. First, if hic vertices v_1 and v_2 share a common neighbor *n*, it could happen that v_1 wants to recolor *n* to a different color than v_2 does. To avoid this first type of conflict, Heuristic 2 creates sets of hic vertices that share a common neighbor and recolors all vertices in the set with the same color. Second, if hic vertex v_1 has a neighbor n_1, hic vertex v_2 has a neighbor n_2, and n_1 is a neighbor of n_2, it could happen that v_1 and v_2 want to recolor n_1 and n_2, respectively, to the same color. Doing so would yield a graph in which the adjacent vertices n_1 and n_2 have the same color. To avoid this second type of conflict, Heuristic 2 removes the available colors for recoloring the neighbors of one hic vertex from the available colors for recoloring the neighbors of the other hic vertex if they have adjacent neighbors. This ensures that the neighbors cannot be recolored with the same color. For example, if one hic vertex is considering recoloring one of its neighbors to blue, removing blue from the available colors for recoloring the neighbors of the other hic vertex guarantees that it will not use blue for any of its neighbors, thus avoiding the conflict. Heuristic 2 always removes the overlapping colors of the larger set from the smaller sets since the smaller set is less likely to be useful.

Figure 9 provides an example of how Heuristic 2 works. Figure 9a shows an LDF colored graph in which vertices I and J have the highest color, i.e., there are two hic vertices. I and J do not share a common neighbor, so Heuristic 2 treats them independently and does not put them in the same set. In other words, the first type of conflict cannot occur. Examining I's neighbors, we find that only vertex C can be recolored (to blue). Examining J's neighbors, we find that both vertices A and E can be recolored (to yellow). To avoid the second type of conflict, Heuristic 2 must ensure that A and D are not recolored to the same value and that C and H are not recolored to the same value. To guarantee that, it removes yellow (A's recoloring set) from D's set of available recoloring colors as well as blue (C's recoloring set) from H's set of available recoloring colors. In both cases, the colors from the larger set with one entry are removed from the smaller set, which is empty in this example. At this point, Heuristic 2 recolors C to blue, which frees up red for I, and either A or E to yellow. Assuming it picks E, this frees up blue for J. The recolored graph is shown in Figure 9b. Note that the heuristic reduced the total number of colors needed by recoloring the two hic vertices with different new colors.

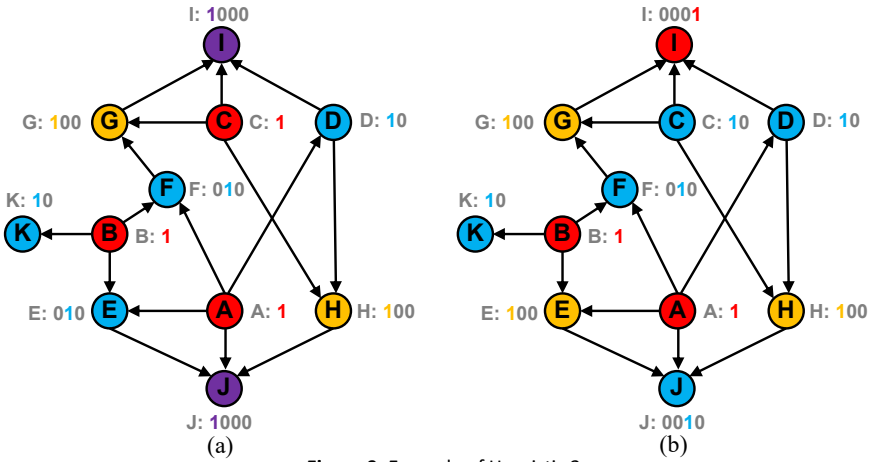


Figure 9: Example of Heuristic 2

For performance and parallelization reasons, we implemented Heuristic 2 as presented in Algorithm 5. It first populates a worklist with all hic vertices (lines 3 through 7). Next, it creates a disjoint-set (union-find) data structure in which each hic vertex is a set of its own (line 8). Then, for each vertex (line 9), it collects all hic neighbors in a set (lines 10 through 15) and merges them in the disjoint-set data structure (line 16) to avoid conflicts of the first type. For each of the resulting disjoint sets, 32 bitmaps are allocated. Each bitmap corresponds to a possible neighbor color and holds the available colors for recoloring all neighbors with that color. The available colors are always higher than the current color and must be less than hic (and less than 32 due to our cap), so the bitmaps are initialized accordingly (line 20). At this point, the heuristic goes over all neighbors *n* (line 24) of each hic vertex *v* (line 23) and removes the color of each neighbor's neighbors from the bitmap corresponding to the color of *n* and belonging to the set of *v* (line 30), but only if the colors are under 32 (lines 26 and 28). To avoid conflicts of the second type, Heuristic 2 goes over all pairs of adjacent vertices (lines 36 and 40) where both vertices have hic neighbors (lines 37 and 41) that belong to different sets (line 44), determines the larger set of available colors for recoloring (line 45) and removes the overlapping colors from the smaller set (lines 46 and 48). Finally, it traverses all hic

vertices (line 55), checks if recoloring any neighbors with a color under 32 is possible (line 59), and, if so, recolors the corresponding neighbors (line 62) and the hic vertex (line 65).

All outer and some inner for-each loops of Algorithm 5 can be executed in parallel. The worklist is populated concurrently using atomicAdd instructions. The union operations and the path-compressing find operations on the disjoint-set data structure are parallelized as described elsewhere [29]. The initialization of the bitmaps is embarrassingly parallel. The unallowed elements are removed from the bitmaps using atomic instructions. This is also how the overlapping elements from a larger conflicting set are removed from the smaller set. The recoloring itself is embarrassingly parallel.

ALGORITHM 5: Color-reduction Heuristic 2

```
1:  hic ← highest used color
2:  worklist ← Ø
3:  for each vertex v in G do
4:      if color_v = hic then
5:          worklist ← worklist ∪ v
6:      end if
7:  end for
8:  disjointsets ← worklist
9:  for each vertex v in G do
10:     set ← Ø
11:     for each neighbor n of v in G do
12:         if color_n = hic then
13:             set ← set ∪ n
14:         end if
15:     end for
16:     merge all elements of set in disjointsets
17: end for
18: for each set s in disjointsets do
19:     for each i form 0 to 31 do
20:         available_s,i ← {i+1, i+2, …, min(hic-1, 31)}
21:     end for
22: end for
23: for each vertex v in worklist do
24:     for each neighbor n of v in G do
25:         c ← color_n
26:         if c < 32 then
27:             for each neighbor k of n in G do
28:                 if color_k < 32 then
29:                     s ← disjointsets(v)
30:                     available_s,c ← available_s,c \ color_k
```

```
31:                     end if
32:                 end for
33:             end if
34:         end for
35: end for
36: for each vertex v in G do
37:     if v has hic neighbor then
38:         c ← color_v
39:         s ← disjointsets(any hic neighbor of v)
40:         for each neighbor n of v in G do
41:             if n has hic neighbor then
42:                 d ← color_n
43:                 t ← disjointsets(any hic neighbor of n)
44:                 if s ≠ t then
45:                     if available_s,c < available_t,d then
46:                         available_s,c ← available_s,c \ available_t,d
47:                     else
48:                         available_t,d ← available_t,d \ available_s,c
49:                     end if
50:                 end if
51:             end if
52:         end for
53:     end if
54: end for
55: for each vertex v in worklist do
56:     s ← disjointsets(v)
57:     for each i form 0 to 31 do
58:         b ← best(available_s,i)
59:         if b ≠ Ø then
60:             for each neighbor n of v in G do
61:                 if color_n = i then
62:                     color_n ← b
63:                 end if
64:             end for
65:             color_v ← i
66:             break
67:         end if
68:     end for
69: end for
```

5 RELATED WORK

A large amount of related work exists on graph coloring. Yet, there is very little prior work on improvement heuristics to reduce the number of colors used and no other work that proposes shortcuts to increase the parallelism.

The classical sequential graph-coloring algorithm is based on the greedy first-fit heuristic. Several other heuristics have been proposed that use relatively few colors and have good bounds on their computational complexity (cf. Section 1). In contrast, parallel algorithms have not been studied as extensively. Nevertheless, there are a few polynomial-time algorithms, some of which can solve the problem using as few colors as the sequential algorithms.

In 1986, Luby designed a Monte Carlo algorithm to find a maximal independent set (MIS) in parallel [32]. All vertices in the MIS are given the same color. Then the algorithm finds a new MIS among the remaining vertices and assigns the vertices in the second MIS the second color, and so on until all vertices have been colored.

In 1993, Mark Jones and Paul Plassmann proposed a new graph coloring heuristic (JP) [30] based on Luby's Monte Carlo algorithm. Luby's algorithm selects new random numbers in each iteration, which requires global synchronization. Moreover, generating the random numbers incurs overhead. Jones and Plassmann largely eliminate the global synchronization and this overhead by choosing a random number for each vertex only once. In other words, their algorithm does not assign new random numbers in each round when a new independent set needs to be calculated but reuses the previously assigned numbers. The unique vertex IDs are used to resolve conflicts if neighboring vertices end up with the same random number. Then the algorithm checks all the neighbors of each uncolored vertex v. If v has the highest random number (i.e., the highest priority) among its uncolored neighbors, the lowest available color is assigned to v. The algorithm repeats the last two steps until all vertices have colors.

The Largest-Degree-First (LDF) heuristic assigns a priority to each vertex that is proportional to the degree of the vertex. This causes the vertices to be colored in decreasing degree order, i.e., the vertices with the highest degree are colored first. Using this ordering typically yields a better coloring quality than the JP and greedy algorithms. Random numbers are used to resolve conflicts when two neighboring vertices have the same degree [26]. The JP algorithm can easily be augmented with LDF. The resulting parallel JP-LDF algorithm is outlined in Section 2.

The Smallest-Degree-Last (SDL) heuristic tries to improve upon the coloring quality of LDF by using more sophisticated weights [34]. It comprises a weighting and a coloring phase. In the weighting phase, the algorithm starts by finding all vertices with the minimum degree d_{min}. These vertices are assigned weights and are removed from the graph, which changes the degree of their neighbors. The algorithm repeatedly removes vertices with degree d_{min} and assigns larger weights in each iteration. Once there are no vertices of degree d_{min} left, the algorithm continues with vertices of degree d_{min}+1 and so on. Then the coloring phase starts with the vertices that have the highest weights. It works in the same way as LDF except is uses the weights instead of the degrees to determine the order in which to color the vertices. As mentioned in Section 1, SDL tends to yield a very good coloring quality but is slow.

In 2011, Grosset et al. implemented their 3-step GM algorithm in CUDA [24]. It partitions the graph, colors each partition independently, and resolves conflicts along the border first on the GPU and then on the CPU using one of the heuristics described in Section 1. The resulting runtime is often worse than the sequential algorithm [8].

The CUSPARSE library [14] includes the "csrcolor" graph-coloring code [7]. As the name implies, it operates on graphs in CSR format. We use the same format in ECL-GC. Csrcolor is based on the Jones-Plassmann and Cohen-Castonguay [9] algorithms. It uses multiple hash functions to generate the "random" numbers for each vertex. The local maximums and minimums of the hash values are employed to produce two distinct maximal independent sets. The GPU implementation is three to four times faster than the JP algorithm. However, csrcolor typically requires over twice as many colors as the JP algorithm.

Chen et al. [8] proposed two graph coloring algorithms based on Nasre's ideas for implementing irregular algorithms on GPUs [36]. The first is a topology-driven algorithm. It uses the first-fit heuristic to color all vertices in parallel with the first permissible color. Conflicts between adjacent vertices with the same color are handled by allowing the vertex with the highest degree to preserve its color whereas the remaining conflicting vertices are uncolored. Chen et al.'s second algorithm works in the same way but is data-driven. It maintains two worklists for holding the vertices that need to be processed, making it more work efficient, but maintaining the worklists incurs overhead.

Chen et al. implemented multiple versions of their algorithms with different optimizations [8], including bitmap operations to reduce the memory footprint and the time consumed in reading and writing the color mask. ECL-GC employs bitmaps for the same reason but also to facilitate the shortcuts. For better load balancing, they implemented Merrill's balancing strategy [35], which maps the workload of a vertex to a thread, warp, or block depending on the size of its neighbor list. Similarly, ECL-GC uses threads for processing vertices with degrees under 32 and warps for higher-degree vertices.

Osama et al. [31] wrote GPU versions of Jones-Plassman's and of Luby's graph-coloring algorithms based on two abstractions, data-centric using the Gunrock framework and linear-algebra-based using GraphBLAS. For Gunrock, they employed three operations: 1) an advance operator to generate a new frontier from the current frontier by visiting the neighbors of the current frontier, 2) a compute operator that performs an operation on all elements in the input frontier, and 3) a neighbor-reduce operator that uses the advance operator to visit the neighbors of each item in the input frontier and performs a segmented reduction over the neighborhood. For Luby's independent-set-based algorithm, they form two independent sets in each iteration. Instead of only assigning vertices with the largest random number relative to their neighbors to a maximum independent color set, they also assign colors to vertices with the smallest random number to a minimum independent color set. Since the max-comparison and min-comparison sets are mutually exclusive, they perform the assignment of two colors in every iteration with no additional overhead. This optimization reduces the coloring time by almost half. They also proposed a Hash Independent Set (IS) algorithm, which is a modification of the Maximal Independent Set algorithm. Each vertex compares only its neighbors with one another and adds the neighbor vertex with the largest random number relative to all neighbors to the hash color set. The Hash IS color set can contain more vertices than the independent color set. However, the color set is not independent because each vertex knows only its local topology, which may cause a conflict. Conflict resolution is another compute operation. It compares all colored vertices with their neighbors in a serial for-loop and, if the resolution detects a conflict, it resets one of the violating vertices to be uncolored. To amortize the cost of the conflict resolution, the implementation uses a hash table to inform the vertex about colors that cannot be used. This implementation yields a fast runtime but not a very good coloring quality.

Deveci et al. [16] proposed a parallel vertex-based (VB) iterative graph coloring algorithm and present two optimizations to enable VB to run efficiently on a GPU. First, they allocate a small FORBIDDEN array of fixed size for each thread. These arrays are the inverse of the possible color lists we use. Since the arrays have a fixed size, they also employ a COLORRANGE for handling more colors than fit in the array. If a color cannot be found in the given range, the adjacency list is traversed again to populate the FORBIDDEN array based on the next COLORRANGE. The second optimization is to eliminate the FORBIDDEN array and use a bitmap instead. Deveci et al. also they present an edge-based coloring algorithm. In this algorithm, they create a list of the forbidden colors of each vertex v and initialize it to $\emptyset$. Then they go over all the vertices, pick the smallest available color for v based on v's list of forbidden

colors, and check for conflicts. If a conflict occurs, they go over all edges and atomically update the list of forbidden colors for all vertices. These last two steps repeat until all vertices are colored without conflict.

Gebremedhin et al. [22] proposed two graph coloring heuristics for CPUs. The first heuristic partitions the vertex set into p successive blocks of equal size. The parallel coloring comprises n/p parallel steps with barriers at the end of each step. In this algorithm, two processors may simultaneously attempt to color vertices in the same block that are adjacent to each other, which may result in an invalid coloring (pseudo coloring). The next step is to check for any conflict and, if a conflict is detected, the edge in the conflict will be stored in a table. The last step is to color all the vertices stored in this table sequentially. For the second algorithm, they modified the first algorithm to use fewer colors and based this improvement on Culberson's Iterated Greedy (IG) coloring heuristic [13]. In the result section, we compare to the GM algorithm, which employs the IG heuristic.

Besta et al. [3] introduced the first graph coloring algorithms with proven theoretical bounds on work, depth, and quality. They introduced three CPU algorithms that use a vertex ordering called Approximate Degeneracy Ordering (ADG) when selecting which vertex to color next. The first algorithm, JP-ADG, is based on Jones-Plassman's algorithm. The second algorithm employs the speculation-based DEC-ADG algorithm where vertices are colored independently using a "speculative coloring". Any coloring conflicts are resolved by repeated coloring attempts. The third algorithm, DEC-ADG-ITR, is based on a recent algorithm called ITR [2]. DEC-ADG-ITR focuses on improving the coloring quality of ITR both in theory and practice. They compared their algorithms to the Jones-Plassman algorithm combined with different ordering heuristics and to the original ITR algorithm. Using both JP-ADG and DEC-ADG-ITR, they were able to improve the coloring quality with a good runtime compared to the baseline. We compare ECL-GC to these algorithms in the result section.

Culberson et al. [13] proposed one of the earliest graph coloring *improvement* heuristics. They use a simple greedy algorithm as the core of an iterative process that permutes the color sets produced by a previous coloring. For example, in the second iteration, all the blue vertices from the first iteration may be colored first, followed by all the red vertices, etc. Applying such a permutation yields a new coloring in which the number of colors is guaranteed not to increase but may decrease, i.e., improve the quality. The premutation is generated based on different reordering heuristics for the color sets, including reverse order, increasing size (processes the smaller color sets first), decreasing size, increasing degree, decreasing degree, and random ordering. Their first approach is to run the greedy algorithm with the increasing size ordering. The resulting coloring is used by the second iteration, which uses the decreasing size ordering. This repeats for multiple iterations. The second approach is to randomly switch between heuristics in each iteration. The algorithm terminates when reaching a specific number of iterations, when there is no improvement for a specified number of iterations, or when a desired number of colors is achieved. The second approach tends to yield better coloring as it breaks up cyclic patterns in the coloring process. Culberson et al.'s improvement heuristics recolor the entire graph repeatedly. In contrast, our improvement heuristics only recolor the vertices with the highest color and their neighbors.

6 EXPERIMENTAL METHODOLOGY

We evaluate the graph-coloring codes listed in Table 2. Some of these programs have multiple versions. We only show results for the fastest version as well as the version requiring the fewest colors if the number of colors is substantially smaller.

In the evaluated codes, we only measured the runtime of the color computation, excluding the time it takes to copy the graphs into main memory, to transfer data to and from the GPU (unless otherwise noted), and to verify the result. We ran each experiment three times and use the best measured runtime. The ECL-GC runtimes only vary by a few percent between runs. For all ECL-GC implementations, we verified the solution by comparing it to that of the serial code in addition to checking that no adjacent vertices have the same color.

We present results from two GPUs. The first is a Volta-based Titan V with 5120 processing elements distributed over 80 multiprocessors. Each multiprocessor has 96 kB of L1 data cache/shared memory. The 80 multiprocessors share a 4.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 652 GB/s. The second GPU is a Turing-based GeForce RTX 2070 Super with 2560 processing elements distributed over 40 multiprocessors. Each multiprocessor has 96 kB of L1 data cache/shared memory. The 40 multiprocessors share a 4 MB L2 cache as well as 8 GB of global memory with a peak bandwidth of 448 GB/s.

Table 2. The graph coloring codes we evaluate

Device	Ser/Par	Name	Version	Source
GPU	Parallel	ECL-GC (our code)	1.0	[18]
		CUSP	0.5.1	[15]
		csrcolor	9.2.88	[7]
		Data-wlc	1.0	[8]
		Data-pq	1.0	[8]
		Gunrock's LoadBalance	1.0	[39]
		kokkos-VB	1.0	[16][31]
CPU	Parallel	GMMP-NT		[12]
		FirstFit	1.0	[8]
		Grappolo		[23]
		kokkos-VB	1.0	[16][31]
		JP-IADG-AVG-IS	1.0	[3]
		DEC-ADG-ITR	1.0	[3]
CPU	Serial	LF-D1		[12]
		FirstFit	1.0	[8]
		Boost	1.66.0	[4]
		kokkos-Serial	1.0	[16][31]

The system we used for the serial and parallel CPU codes has dual 10-core 3.1 GHz Xeon E5-2687W v3 CPUs. Hyperthreading is enabled, i.e., the 20 cores can simultaneously run 40 threads. Each core has separate 32 kB L1 caches, a 256 kB L2 cache, and the cores on a socket share a 25 MB L3 cache. The 128 GB main memory has a peak bandwidth of 68 GB/s. The operating system is Fedora 23.

We compiled all GPU codes with nvcc 9.2 using "-O3 -arch=sm_70" for the Titan V and "-O3 -arch=sm_75" for the GeForce RTX 2070 Super. The CPU codes were compiled with gcc/g++ 7.3.1 using "-O3 -march=native".

Table 3. Information about the input graphs

Graph name	Type	Origin	Vertices	Edges	d_{avg}	d_{max}	$d \geq 32$
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	4.0	4	0.0%
amazon0601	co-purchases	SNAP	403,394	4,886,816	12.1	2,752	3.3%
as-skitter	Internet topo.	SNAP	1,696,415	22,190,596	13.1	35,455	6.3%
citationCiteseer	publication	SMC	268,495	2,313,294	8.6	1,318	3.6%
cit-Patents	patent cites	SMC	3,774,768	33,037,894	8.8	793	3.0%
coPapersDBLP	publication	SMC	540,486	30,491,458	56.4	3,299	52.5%
delaunay_n24	triangulation	SMC	16,777,216	100,663,202	6.0	26	0.0%
europe_osm	road map	SMC	50,912,018	108,109,320	2.1	13	0.0%
in-2004	web links	SMC	1,382,908	27,182,946	19.7	21,869	8.4%
internet	Internet topo.	SMC	124,651	387,240	3.1	151	0.3%
kron_g500-logn21	Kronecker	SMC	2,097,152	182,081,864	86.8	213,904	19.3%
r4-2e23.sym	random	Galois	8,388,608	67,108,846	8.0	26	0.0%
rmat16.sym	RMAT	Galois	65,536	967,866	14.8	569	11.4%
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	15.7	3,687	12.4%
soc-LiveJournal1	community	SNAP	4,847,571	85,702,474	17.7	20,333	14.0%
uk-2002	web links	SMC	18,520,486	523,574,516	28.3	194,955	18.6%
USA-road-d.NY	road map	Dimacs	264,346	730,100	2.8	8	0.0%
USA-road-d.USA	road map	Dimacs	23,947,347	57,708,624	2.4	9	0.0%

We used the 18 graphs listed in Table 3 as inputs. They were obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dimacs) [17], the Galois framework (Galois) [19], the Stanford Network Analysis Platform (SNAP) [42], and the SuiteSparse Matrix Collection (SMC) [43]. The table lists the name, type, source, number of vertices, number of edges, average degree, maximum degree, and the percentage of vertices with a degree of at least 32 (for which we use simplified shortcuts). Where necessary, we made the graphs undirected and removed self-edges. Due to the CSR format, each undirected edge is represented by two directed edges. While it may or may not make sense to color these graphs, we selected them for their wide variety.

7 GRAPH-COLORING RESULTS

In this section, we first study the amount of parallelism. Second, we evaluate the coloring quality. Third, we investigate the throughput in completed vertices per second, that is, the number of vertices divided by the runtime. The improvement heuristics to reduce the number of colors used are studied in the next section.

7.1 Amount of Parallelism

In this subsection, we evaluate the intrinsic amount of parallelism with and without the shortcuts. We express the parallelism as the number of vertices divided by the number of steps it takes to color a graph in an architecture-agnostic way, i.e., assuming a machine with infinite resources that processes as many vertices per step as possible subject only to data dependencies. Hence, in every step, all vertices are colored that do not have to wait for uncolored higher-priority neighbors.

Figures 10 and 11 show the steps along the x axis and how many vertices are colored per step along the y axis. Note that the y axes use a logarithmic scale to better show what happens in the last steps, but this upsets certain intuitions that would hold if a linear scale were used, such as that both curves enclose the same area. The larger the number of colored vertices in each step the higher the amount of parallelism is. The blue curve shows the results

without the shortcuts and the red curve with the shortcuts. Both approaches yield identical colorings and perform the same amount of total work. Therefore, finishing in fewer steps implies a higher average parallelism.

Figure 10 shows that the shortcuts can yield a large increase in parallelism, in this case a 7.85-fold increase. In contrast, Figure 11 shows the worst case, i.e., an example where the shortcuts do not increase the average parallelism because the two tails overlap and both end in the same step. However, the shortcuts significantly increase the average parallelism on most of the tested inputs as shown in Table 4, which lists the number of steps, the average parallelism, and the improvement in parallelism for all 18 graphs.

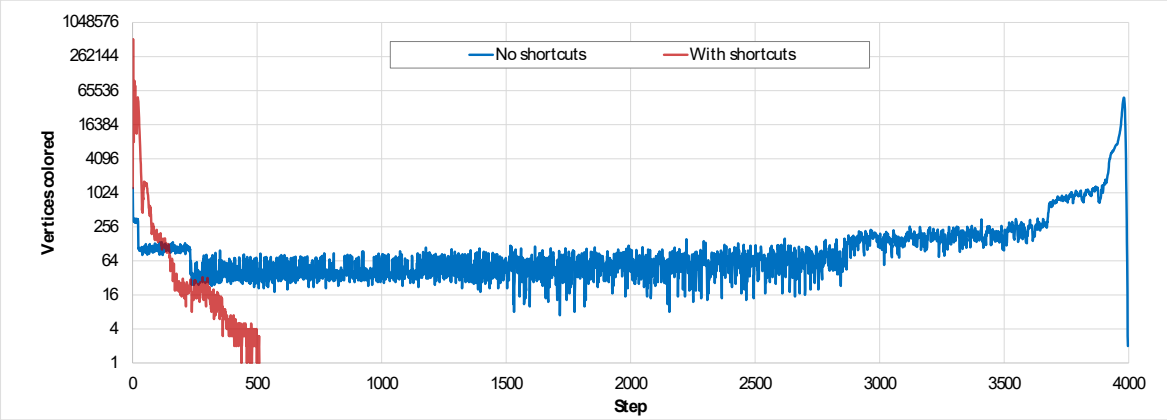


Figure 10: Amount of parallelism in each step on the kron_g500-logn21 graph

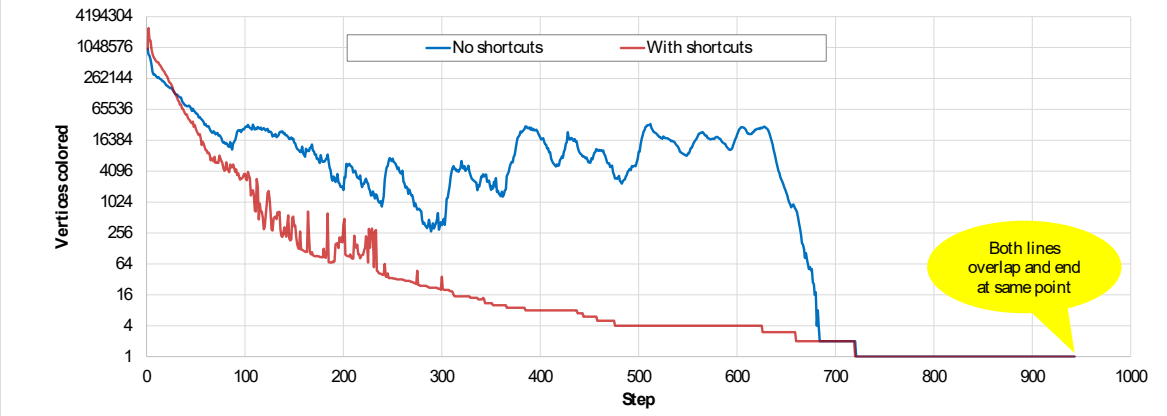


Figure 11: Amount of parallelism in each step on the uk-2002 graph

Table 4: Number of steps and average amount of parallelism with and without the shortcuts

Graph	Steps w/o shortcuts	Steps with shortcuts	Avg parallelism w/o shortcuts	Avg parallelism with shortcuts	Increase in parallelism
2d-2e20.sym	14	12	74,898.3	87,381.3	1.17
amazon0601	55	24	7,334.4	16,808.1	2.29
as-skitter	481	73	3,526.9	23,238.6	6.59
citationCiteseer	67	20	4,007.4	13,424.8	3.35
cit-Patents	140	26	26,962.6	145,183.4	5.38
coPapersDBLP	802	338	673.9	1,599.1	2.37
delaunay_n24	25	17	671,088.6	986,895.1	1.47
europe_osm	13	11	3,916,309.1	4,628,365.3	1.18
in-2004	501	490	2,760.3	2,822.3	1.02
internet	27	13	4,616.7	9,588.5	2.08
kron_g500-logn21	3,997	509	524.7	4,120.1	7.85
r4-2e23.sym	30	17	279,620.3	493,447.5	1.76
rmat16.sym	188	30	348.6	2,184.5	6.27
rmat22.sym	644	52	6,512.9	80,659.7	12.38
soc-LiveJournal1	1,095	322	4,427.0	15,054.6	3.40
uk-2002	943	943	19,640.0	19,640.0	1.00
USA-road-d.NY	12	10	22,028.8	26,434.6	1.20
USA-road-d.USA	14	13	1,710,524.8	1,842,103.6	1.08

In the worst case (uk-2002), the amount of parallelism does not increase. This only happens on 1 of the 18 tested graphs. In the best case (rmat22.sym), the parallelism is over 12 times higher. Based on the geometric mean, it is 2.5 times higher, demonstrating the potential of the shortcuts.

Figure 12 shows the fraction of the vertices that is colored during initialization (blue), using the shortcuts (green), and conventionally (red), i.e., after all higher-priority neighbors have been colored. On average, 52.6% of the vertices are colored conventionally, 38.8% are colored using the shortcuts, and 8.6% are colored during initialization. The number of vertices colored in the initialization phase reflects the number of roots in the DAG. Since the shortcuts shorten the dependence chains, they tend to be more effective, i.e., color a larger fraction of the vertices, on graphs with larger average degrees like kron_g500-logn21, which has a high maximum and average degree.

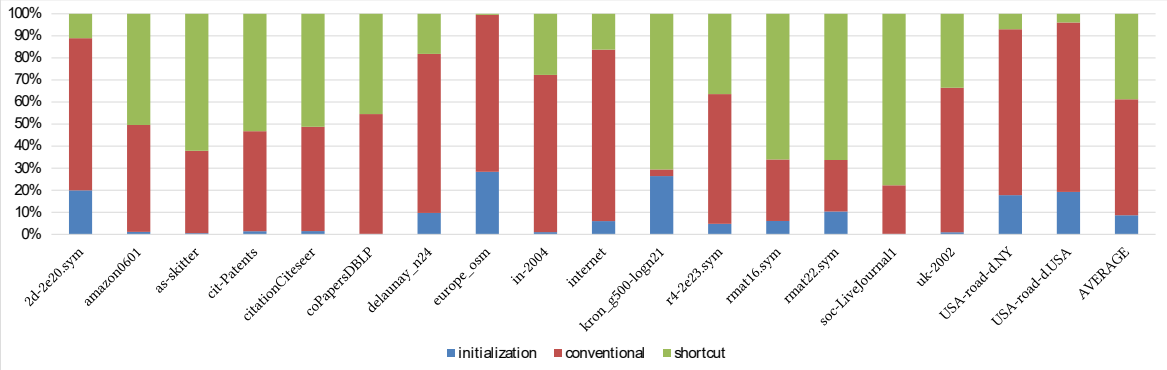


Figure 12: Fraction of colors assigned by various means

7.2 Comparison with GPU Codes

This subsection compares the performance of ECL-GC (with the shortcuts) and ECL-GC_Reduction (with the shortcuts as well as the two color-reduction heuristics) to that of leading GPU graph-coloring codes from the literature. We compare with CUSP, csrcolor, Data-wlc and Data-pq, the two fastest versions of Chen et al.'s algorithms described in Section 4, Gunrock's LoadBalance algorithm, and Kokkos-kernels vertex-based (VB) algorithm. Gunrock includes several different algorithms. We selected the LoadBalance algorithm because it yields the best coloring quality and is the second fastest of their algorithms. Their fastest algorithm yields the worst coloring quality. We were unable to run LoadBalance with the uk-2002 input, which may be due to the large size of this graph. Kokkos-kernels [16][31] also includes several parallel algorithms, and we selected the vertex-based (Kokkos-VB) algorithm because it is their fastest algorithm. This algorithm is based on Deveci et al.'s [16] vertex-based algorithm.

7.2.1 Coloring Quality

Figure 13 shows the number of colors needed by the seven GPU codes. Lower numbers are better. The x axis lists the input graphs and the y axis the number of colors using a logarithmic scale. The rightmost set of bars reflects the geometric mean over all inputs (excluding uk-2002 for LoadBalance).

Both versions of ECL-GC, CUSP, and csrcolor are deterministic and always produce the same coloring for a given input. This is not the case for Data-wlc, Data-pq, and LoadBalance, where the number of colors may vary. For these codes, we show the lowest observed number of colors out of 100 runs on the Titan V. Kokkos-VB is also not deterministic, but we could only run it 3 times and show the lowest observed number of colors.

ECL-GC either uses the smallest or the same number of colors for all inputs compared to the six GPU codes from the literature. LoadBalance and csrcolor require substantially more colors on each graph compared to the other codes. By design, the coloring of ECL-GC is that of JP with LDF, which tends to produce a good coloring quality. As discussed in Section 4, csrcolor requires more colors because it is based on the Cohen-Castonguay algorithm. Data-wlc and Data-pq are based on FirstFit, which typically results in good coloring when paired with LDF. Kokkos-VB produces a coloring quality similar to Data-wlc, but Kokkos-VB requires on average one more color than Data-wlc and 8.9 and 10.7 more colors than our ECL-GC and ECL-GC_Reduction algorithms, respectively. ECL-GC_Reduction lowers the number of colors on half of the 18 inputs relative to ECL-GC, highlighting the benefit of the color-reduction heuristics. For example, it reduces the number from five to four on the three road maps. It exceeds the solution quality of the other codes from the literature on 14 of the 18 inputs and is tied on the remaining four graphs. The geometric mean is 30.6 colors for ECL-GC, 28.8 for ECL-GC_Reduction, 37.2 for Data-wlc, 34.3 colors for Data-pq, 149.4 for csrcolor, 35.0 for CUSP, 67.3 for LoadBalance, and 39.5 for Kokkos-VB.

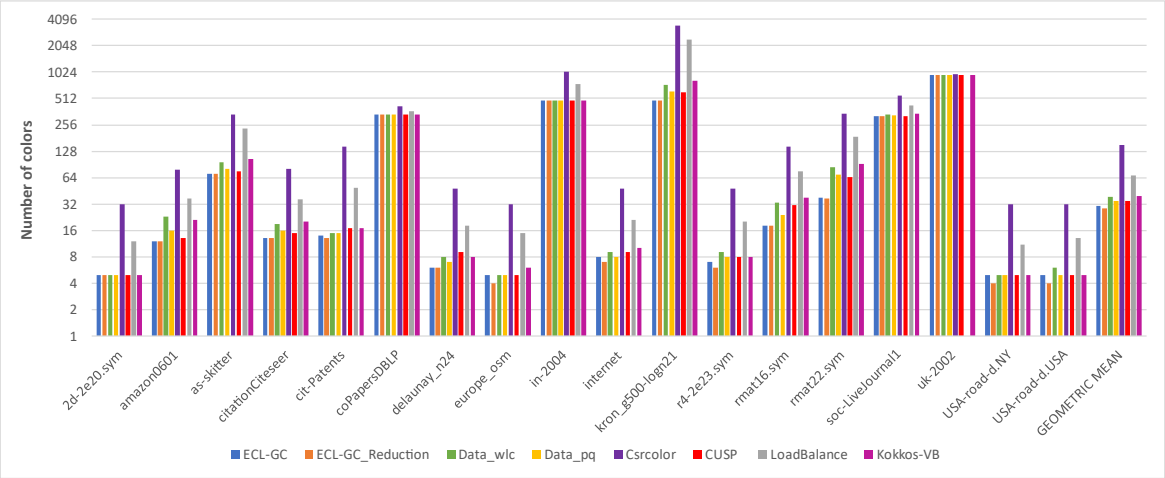


Figure 13: Number of colors needed by the GPU codes

7.2.2 Throughput

Figures 14 and 15 present the throughput of these codes on two different GPU architectures. The x axis lists the inputs and the geometric mean whereas the y axis shows the throughput in millions of completed vertices per second on a logarithmic scale. Throughput is a higher-is-better metric.

Figure 14 shows the throughput on the Titan V. ECL-GC, i.e., our implementation with the shortcuts, is faster than CUSP and LoadBalance on all tested inputs. It is faster than the remaining codes from the literature on 14 of the 18 graphs. Note that, in each case where the other codes are faster, they require more colors. Based on the geometric mean, ECL-GC is 3.7 times faster than Data-wlc, 2.9 times faster than Data-pq, 5.5 times faster than csrcolor, 29.9 times faster than CUSP, 28.8 times faster than LoadBalance and 4.1 times faster than Kokkos-VB.

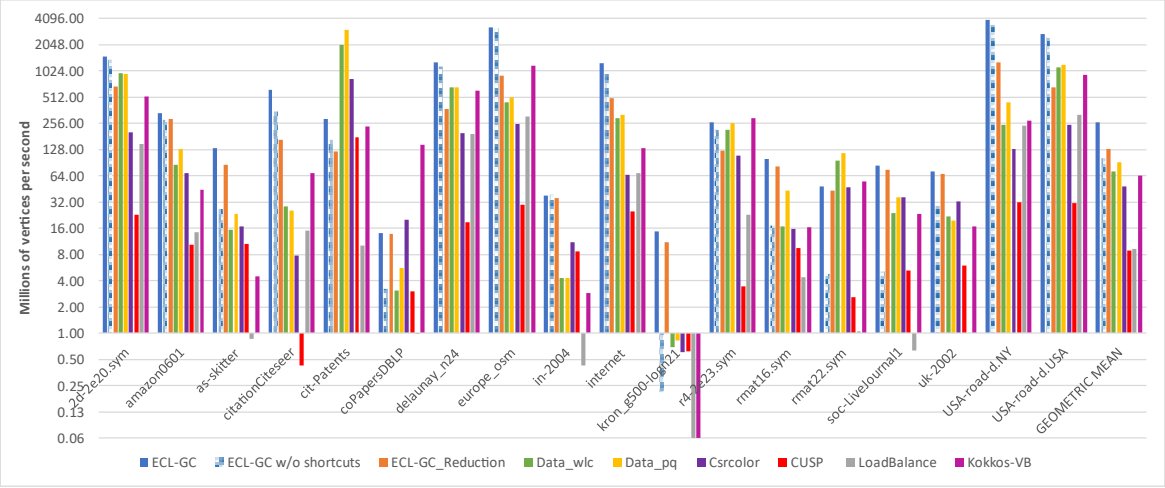


Figure 14: Throughput in millions of completed vertices per second on a Titan V

ECL-GC_Reduction is half as fast as ECL-GC, meaning that including the color-reduction heuristics doubles the mean runtime. However, ECL-GC_Reduction is still faster than the codes from the literature on 10 of the 18 inputs and has a higher geometric-mean throughput. These results show that the heuristics are relatively fast.

We correlated the speedup of ECL-GC over the other codes with various graph properties and found a moderate correlation with both the maximum and the average degree, which is expected because the higher the degree the higher the chance that a vertex must wait for higher-priority neighbors, which is where the shortcuts can help. On the kron_g500-logn21 graph, which has the highest average and maximum degree of the graphs listed in Table 3, ECL-GC is 17.8 times faster than Data-pq, the second fastest of the GPU codes. Due to its high degree, this graph requires the most work per vertex, which is why it results in a low throughput for all tested codes.

For reference, Figure 14 also shows results for "ECL-GC w/o shortcuts", which is ECL-GC with the shortcuts disabled. Its geometric-mean performance is slightly higher than that of the other codes, meaning our baseline implementation performs on par with the best codes from the literature. When enabling the shortcuts, our implementation becomes 2.6 times faster. This speedup demonstrates the usefulness of the shortcuts in practice. The next section discusses the performance of the two shortcuts in more detail.

Figure 15 shows the throughput results for the RTX 2070 Super. CUSP does not run on this newer GPU. ECL-GC outperforms LoadBalance on all tested inputs. It outperforms Data-pq and Data-wlc on all but one, csrcolor on 16 of the 18 graphs, and Kokkos-VB on 14 of the 18 graphs. In all those cases, ECL-GC uses substantially fewer colors. Including the color-reduction heuristics results in a slowdown of 1.6 on this GPU, making ECL-GC_Reduction faster than Kokkos-VB and Csrcolor on 11 and 17 of the tested inputs, respectively. However, in 5 of the 7 cases where the reduction code is slower than Kokkos-VB it uses substantially fewer colors. Based on the geometric mean, ECL-GC is 5.4 times faster than Data-wlc, 4.9 times faster than Data-pq, 4.2 times faster than csrcolor, 26.4 times faster than LoadBalance, and 3.6 times faster than Kokkos-VB.

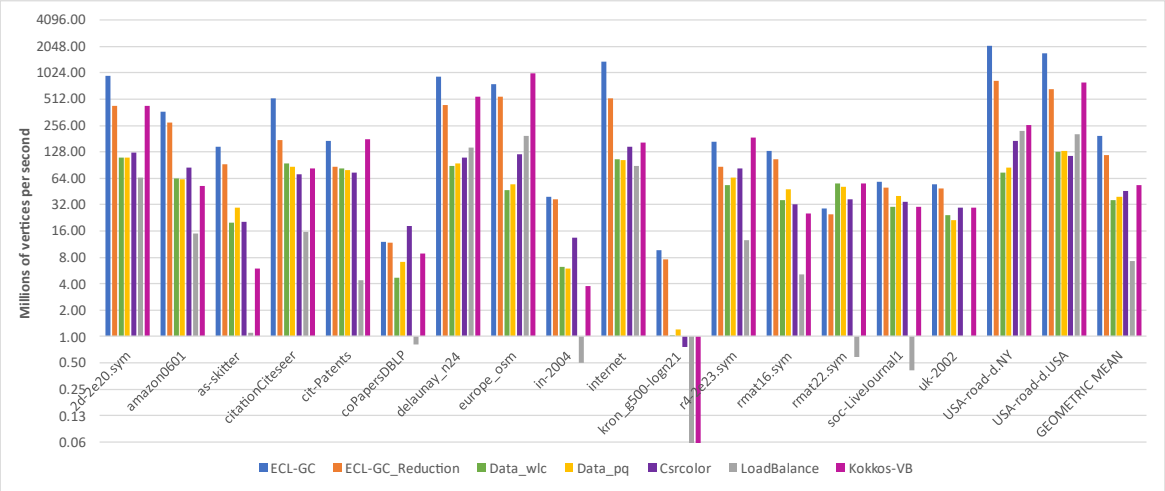


Figure 15: Throughput in millions of completed vertices per second on an RTX 2070 Super

7.2.3 Shortcut Performance

Table 5 presents the performance benefit due to the shortcuts on the Titan V. It shows the speedups attained when using only Shortcut 1 (+SC1), only Shortcut 2 (+SC2), and both shortcuts together (+SC1+SC2) relative to our code without any shortcuts (baseline).

On all tested inputs, using both shortcuts together is always faster than using no shortcut. In the worst case, the shortcuts only improve performance by a factor of 1.027, in the best case by over a factor of 70, and in the mean by a factor of 2.63. These self-relative speedups demonstrate the practical utility of the shortcuts.

Shortcut 1 provides most of the benefit. Adding it never hurts on the tested inputs, helps by a factor of over 2.5 in the mean and by more than a factor of 70 in the best case. Its benefit correlates with the average degree of the graph (r = 0.82), which is why it helps the most on kron_g500-logn21, our highest-degree graph.

Interestingly, adding Shortcut 2 on top of Shortcut 1 hurts in three cases (by up to 2%) and adding it on top of the baseline also hurts in three cases (by up to 1.1%). In the mean, adding Shortcut 2 helps by a few percent and, in the best case, by 25.8%. There are two primary reasons for why Shortcut 2 is not more effective. First, our implementation does not use it on vertices of degree ≥ 32 (under 20% of the vertices in all but one graph, cf. Table 3). Second, employing it does not reduce the number of steps needed until a vertex can be colored. It only makes later steps a little faster because they may be able to skip checking a few neighbors.

Executing the shortcut code itself incurs overhead. If this overhead cannot be amortized, there is a net slowdown, which explains the few cases were adding Shortcut 2 lowers the performance. Fortunately, the benefit of either shortcut is typically high enough to more than amortize this overhead, thus leading to speedups.

Table 5: Speedup on the Titan V due to the shortcuts relative to the baseline code without any shortcuts

input	baseline	+SC1	+SC2	+SC1+SC2
2d-2e20.sym	1.000	1.046	1.005	1.092
amazon0601	1.000	1.236	1.075	1.285
as-skitter	1.000	3.957	1.001	4.198
citationCiteseer	1.000	1.751	1.057	1.816
cit-Patents	1.000	2.015	1.258	2.123
coPapersDBLP	1.000	4.410	1.004	4.407
delaunay_n24	1.000	1.126	1.037	1.168
europe_osm	1.000	1.025	0.999	1.028
in-2004	1.000	1.051	1.019	1.030
internet	1.000	1.248	1.016	1.284
kron_g500-logn21	1.000	70.378	1.004	70.179
r4-2e23.sym	1.000	1.250	1.110	1.339
rmat16.sym	1.000	5.112	1.008	5.251
rmat22.sym	1.000	9.958	0.989	10.163
soc-LiveJournal1	1.000	16.026	0.996	16.028
uk-2002	1.000	2.590	1.010	2.612
USA-road-d.NY	1.000	1.000	1.014	1.027
USA-road-d.USA	1.000	1.068	1.003	1.073
geo mean	1.000	2.570	1.032	2.632

7.3 Comparison with CPU Codes

In the following subsections, we compare the performance of ECL-GC and ECL-GC_Reduction running on the Titan V to that of leading parallel and serial CPU codes. Figures 16 and 18 show the number of colors. The x axis lists the inputs and the geometric mean, and the y axis lists the number of colors using a logarithmic scale. Figures 17 and 19 show the throughput. The x axis again lists the inputs and the geometric mean, and the y axis lists the throughput in completed vertices per second on a logarithmic scale.

Even though ECL-GC was designed to be a fast implementation for GPUs, it is important to compare its performance to the best parallel and serial CPU codes because of the significant tradeoffs between the coloring quality and the execution time. Moreover, graph coloring may suffer from load imbalance and low parallelism depending on the input, which may cause a GPU implementation to be slower than a CPU implementation, especially a parallel CPU implementation.

7.3.1 Parallel CPU Performance Comparison

This subsection compares the throughput and coloring quality to leading parallel CPU codes. We show results for ColPack's GMMP algorithm with the natural (NT) heuristic priority **Error! Reference source not found.**[21], the FirstFit implementation by Chen et al. [8], the graph-coloring code Grappolo [23], Kokkos-kernels' vertex-based algorithm Kokkos-VB [31], and Besta et al.'s [3] algorithms JP-IADG-AVG-IS and DEC-ADG-ITR. We decided to use both JP-IADG-AVG-IS and DEC-ADG-ITR because one produces a better quality result and the other has better performance.

Figure 16 shows the number of colors assigned by the parallel CPU codes and by ECL-GC and ECL-GC_Reduction. As the number of colors may vary from run to run for GMMP-NT, FirstFit, Grappolo, and Kokkos-VB, we present the minimum number observed. Note that some of these codes employ different ordering heuristics and, as such, are not expected to yield the same number of colors. ECL-GC uses fewer colors than ColPack's GMMP-NT and Kokkos-VB on all tested inputs. It uses the smallest or the same number of colors as the FirstFit and Grappolo codes on 11 of the 18 inputs. On the remaining seven inputs, those two codes require one fewer color than ECL-GC's LDF heuristic. Also, ECL-GC uses the smallest or the same number of colors as the JP-IADG-AVG-IS and DEC-ADG-ITR codes on 11 and 12 of the 18 inputs, respectively. The geometric mean is 30.6 colors for ECL-GC, 28.8 for ECL-GC_Reduction, 36 for GMMP-NT, 34.3 for FirstFit, 34 for Grappolo, 36.3 for Kokkos-VB, 29.2 for JP-IADG-AVG-IS, and 29 for DEC-ADG-ITR.

Since ECL-GC_Reduction never uses more colors than ECL-GC, it also requires fewer colors than ColPack's GMMP-NT on all tested inputs. Moreover, it uses the smallest or the same number of colors as the FirstFit and Grappolo codes on 14 and 15 of the 18 inputs, respectively. On the remaining inputs, those two codes require one fewer color than ECL-GC_Reduction, showing the limitation of the heuristics. Also, ECL-GC_Reduction uses the smallest or the same number of colors as JP-IADG-AVG-IS and DEC-ADG-ITR on 14 and 16 of the 18 inputs, respectively. On the remaining inputs, the two codes require one fewer color except on 2d-2e20.sym and kron_g500-logn21, where JP-IADG-AVG-IS requires 3 and 5 fewer colors, respectively. On these two inputs, our code is 1274 and 18.4 times faster than as JP-IADG-AVG-IS.

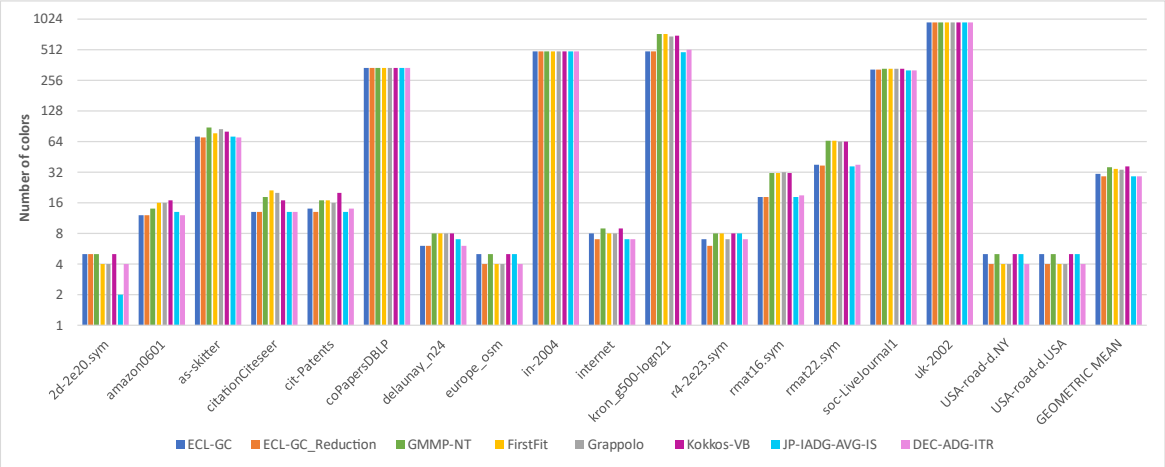


Figure 16: Number of colors needed by the parallel CPU codes as well as by ECL-GC

Figure 17 shows the throughput of the parallel CPU codes on the dual 10-core Xeon system. We ran the codes using both 20 and 40 threads. The results in the figure are for 40 threads since hyperthreading yields a shorter runtime in most cases. ECL-GC running on the Titan V is faster than Grappolo, Kokkos-VB, JP-IADG-AVG-IS, and DEC-ADG-ITR on all tested inputs, faster than GMMP-NT on all but two inputs, and faster than FirstFit on 15 of the 18 inputs. ECL-GC_Reduction is faster than Grappolo, Kokkos-VB, JP-IADG-AVG-IS, and DEC-ADG-ITR on all tested inputs, faster than GMMP-NT on all but three inputs, and faster than FirstFit on all but four inputs. Based on the geometric mean, ECL-GC is 7.2 times faster than GMMP-NT, 4 times faster than FirstFit, 7.8 times faster than Grappolo, 14.9 times faster than Kokkos-VB, 65.7 times faster than JP-IADG-AVG-IS, and 36 times faster than DEC-ADG-ITR on the tested graphs. ECL-GC_Reduction is 3.9 times faster than GMMP-NT, 2.2 times faster than FirstFit, 4.2 times faster than Grappolo, 8.0 times faster than Kokkos-VB, 35.3 times faster than JP-IADG-AVG-IS, and 19.4 times faster than DEC-ADG-ITR

For reference, Figure 17 also shows results for “ECL-GC with CPU/GPU transfer”, which include the time to send the graph to the GPU and the resulting color information back to the CPU. This lowers the geometric-mean throughput by a factor of 2.8, meaning it takes longer to transfer the data than to compute the coloring. Nevertheless, on most of the inputs and in the mean, the throughput is still higher than that of the parallel CPU codes. Of course, this depends on the performance ratio between the CPU and the GPU as well as the speed of the link between the two devices. On our system and graphs, it is often faster to send the data to the GPU, perform the coloring there, and send the result back than to perform the coloring on the CPU. Note that graph coloring is generally only one step in a larger computation. If the previous and next steps are also executed on the GPU, no data transfers are needed.

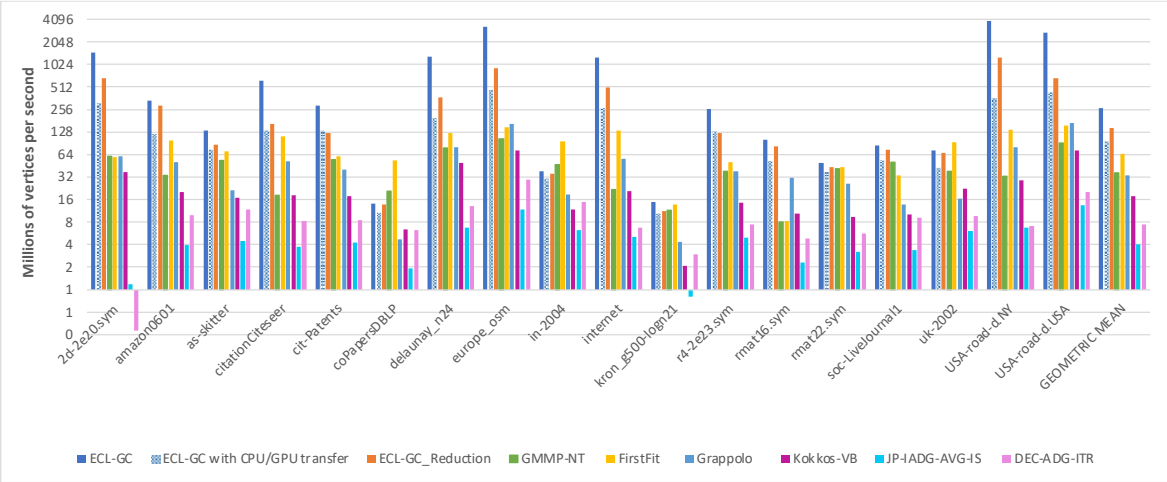


Figure 17: Throughput in millions of completed vertices per second on 20 Xeon cores (Titan V for ECL-GC)

7.3.2 Serial CPU Performance Comparison

This subsection compares the throughput and coloring quality to leading serial codes. We show results for ColPack's sequential graph coloring code with LF ordering and its fastest heuristic (D1) [12], the serial FirstFit code by Chen et al. [8], the graph-coloring code in the Boost library [4] [40], and Kokkos-Serial [31].

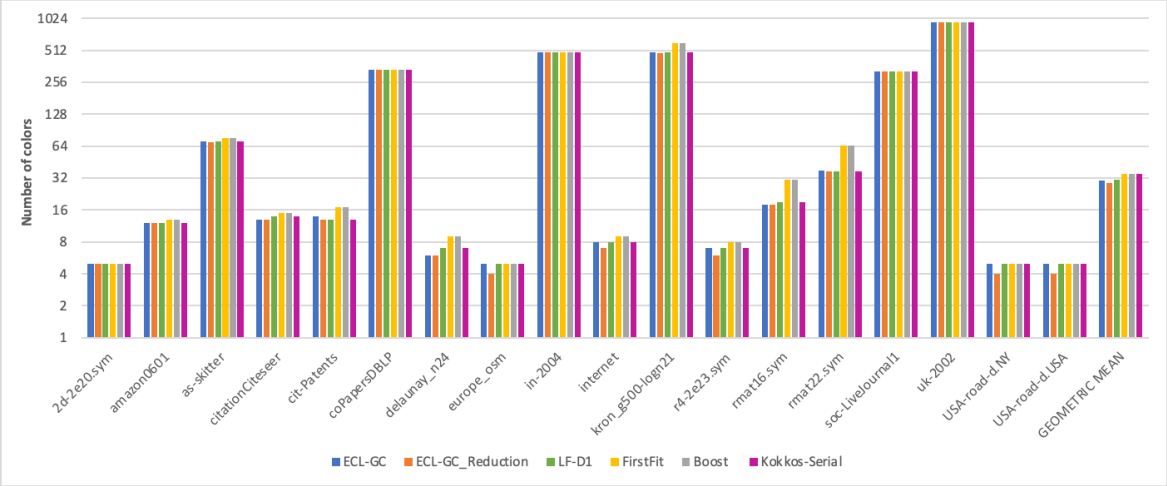


Figure 18: Number of colors needed by the serial CPU codes as well as by ECL-GC

Figure 18 presents the number of colors assigned by the serial codes and by ECL-GC and ECL-GC_Reduction. ECL-GC uses fewer or the same number of colors as serial FirstFit, Boost and Kokkos-Serial on all tested inputs. ECL-GC's

and LF-D1's coloring quality is almost identical. This is expected given that LF-D1 and ECL-GC both employ the largest-degree-first heuristic. The small discrepancies are due to different tie breakers, making LF-D1 use one fewer color than ECL-GC on two graphs and ECL-GC use one fewer color than LF-D1 on four graphs. ECL-GC_Reduction uses the same number or fewer colors than any of the tested serial codes. On 11 of the 18 inputs, it uses fewer colors than any of the serial codes. The geometric mean is 30.6 colors for ECL-GC, 28.8 for ECL-GC_Reduction, 30.9 colors for LF-D1, and 35 colors for FirstFit, Boost, and Kokkos-Serial.

Figure 19 shows the serial throughput on the Xeon system as well as that of ECL-GC and ECL_GC_Reduction running on the Titan V. Both ECL-GC and ECL_GC_Reduction are faster than LF-D1, Boost, and Kokkos-Serial on all inputs and faster than FirstFit on all but two inputs. In those two cases, FirstFit uses more colors. Based on the geometric mean, ECL-GC is 42.9 times faster than LF-D1, 13.2 times faster than FirstFit, 324 times faster than Boost, and 49.4 times faster than Kokkos-Serial. ECL-GC_Reduction is 23.1 times faster than LF-D1, 7.1 times faster than FirstFit, 173.7 times faster than Boost, and 26.5 times faster than Kokkos-Serial.

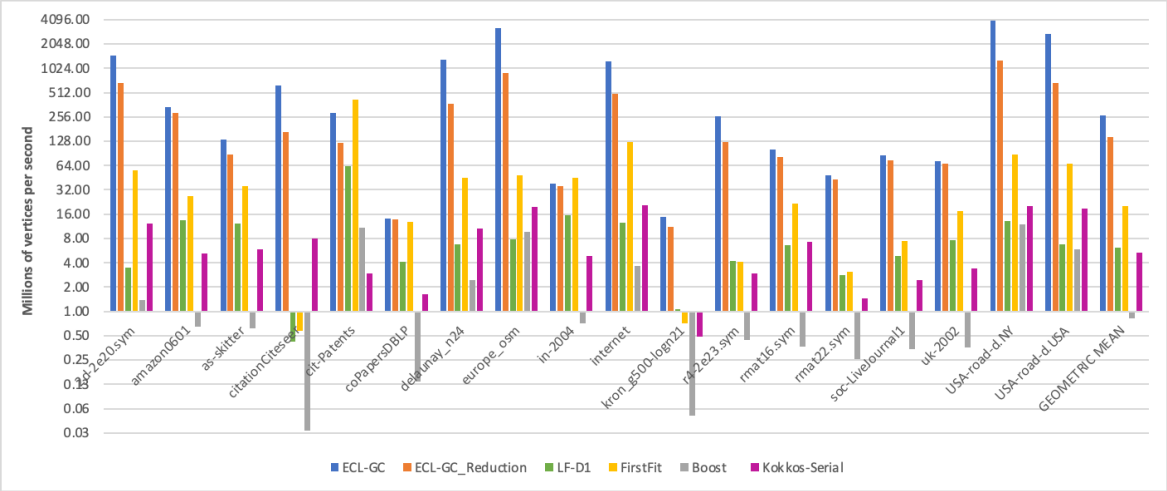


Figure 19: Throughput in millions of completed vertices per second on a Xeon core (Titan V for ECL-GC)

8 SUMMARY AND CONCLUSIONS

Graph coloring is an assignment of colors to the vertices of a graph such that no two adjacent vertices have the same color. It is an important step in many applications and is used, for example, in data mining, image processing, networking, resource allocation, and process scheduling.

We present a deterministic parallel graph-coloring approach that improves upon the Jones-Plassmann algorithm with the largest-degree-first heuristic. It incorporates new algorithmic optimizations called "shortcuts" to increase the parallelism (by 2.5 times based on the geometric mean). Under certain conditions, these shortcuts enable the code to non-speculatively break data dependencies without changing the ultimate color assignment. The shortcuts leverage intermediate coloring information from neighboring vertices, which sometimes allows to correctly color a vertex even before all its higher-priority neighbors have been colored. The shortcuts are particularly useful on high-degree vertices. The paper also presents optimizations to efficiently implement these shortcuts.

We also present two fast and deterministic parallel color-reduction heuristics, one for high- and one low-degree graphs, that improve the coloring quality of ECL-GC on half of the 18 tested graphs by up to 20%. Improving the coloring quality is important in applications like networking and resource allocation, where the number of colors used is critical. Even with the color-reduction heuristics included, ECL-GC is faster on average and on most of the tested inputs than the best GPU and parallel CPU codes from the literature while, at the same time, requiring fewer or the same number of colors on most of the tested input graphs.

We implemented our approaches in CUDA. The code is available at https://cs.txstate.edu/~burtscher/research/ECL-GC/. Running on a Titan V, it is 2.9 times faster (geometric mean) than the fastest prior GPU code, 4.0 times faster than the fastest OpenMP code running with 40 threads on 20 Xeon cores, and 13 times faster than the fastest serial code we could find. Of course, these speedups are system dependent. Our code uses as few or fewer colors as the best GPU codes. Whereas there are a few inputs on which other GPU codes outperform ours in throughput, they require more colors in those cases.

In conclusion, we hope our work will help improve the performance of many applications that incorporate graph coloring as an algorithmic step. Perhaps our ideas will inspire other researchers to develop similar shortcuts to increase the amount of parallelism in other important graph algorithms. Whereas we introduced some of the first graph coloring improvement heuristics in this paper, many more undoubtedly exist and will hopefully soon be researched.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards #1406304 and #1955367, by the Department of Energy, National Nuclear Security Administration under award #DE-NA0003969, and by equipment donations from NVIDIA Corporation.

REFERENCES

[1] Alabandi, Ghadeer, Evan Powers, and Martin Burtscher. Increasing the Parallelism of Graph Coloring via Shortcutting. Proceedings of the 2020 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 262-275. February 2020.

[2] Applegate, David L., Robert E. Bixby, Vašek Chvátal, and William J. Cook. The traveling salesman problem. Princeton university press, 2011.

[3] Besta, Maciej, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. "High-performance parallel graph coloring with strong guarantees on work, depth, and quality." In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-17. IEEE, 2020.

[4] Boost, https://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/index.html, last accessed on 5/6/2021.

[5] Çatalyürek, Ümit V., John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. "Graph coloring algorithms for multi-core and massively multithreaded architectures." *Parallel Computing* 38, no. 10-11 (2012): 576-594.

[6] Çatalyürek, Ümit V., John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. "Graph coloring algorithms for multi-core and massively multithreaded architectures." *Parallel Computing* 38, no. 10-11 (2012): 576-594.

[7] Chen and Li, https://github.com/chenxuhao/csrcolor, last accessed on 5/6/2021.

[8] Chen, Xuhao, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang, and Canqun Yang. "Efficient and high-quality sparse graph coloring on GPUs." *Concurrency and Computation: Practice and Experience* 29, no. 10 (2017): e4064.

[9] Cohen, Jonathan and Patrice Castonguay. "Efficient graph matching and coloring on the GPU." In *GPU Technology Conference*, pp. 1-10. 2012.

[10] Coleman, Thomas F. and Arun Verma. "The efficient computation of sparse Jacobian matrices using automatic differentiation." *SIAM Journal on Scientific Computing* 19, no. 4 (1998): 1210-1233.

[11] Coleman, Thomas F., and Jorge J. Moré. "Estimation of sparse Hessian matrices and graph coloring problems." *Mathematical programming* 28, no. 3 (1984): 243-270.

[12] ColPack, Combinatorial Scientific Computing and Petascale Simulations, https://github.com/CSCsw/ColPack, last accessed on 02/07/2022.

[13] Culberson, Joseph. "Iterated greedy graph coloring and the difficulty landscape." (1992).

[14] Cusparse library. *NVIDIA Corporation*, Santa Clara, California. 2014.

[15] Dalton, S., and N. Bell. "CUSP: A C++ templated sparse matrix library." http://cusplibrary.github.io, last accessed on 5/6/2021.

[16] Deveci, Mehmet, Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. "Parallel graph coloring for manycore architectures." In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 892-901. IEEE, 2016.

[17] DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, http://www.dis.uniroma1.it/challenge9/download.shtml, last accessed on 5/6/2021.

[18] ECL-GC, Texas State University, https://cs.txstate.edu/~burtscher/research/ECL-GC/, last accessed on 5/6/2021.
[19] Galois, ISS - The University of Texas at Austin, https://iss.oden.utexas.edu/?p=projects/galois, last accessed on 5/6/2021.
[20] Garey, Michael R., and David S. Johnson. “Computers and Intractability”, vol. 29. W. H. Freeman and Company, New York (2002): 1-99.
[21] Gebremedhin, Assefaw H., Duc Nguyen, Mostofa Ali Patwary, and Alex Pothen. “ColPack: Graph coloring software for derivative computation and beyond.” *ACM Transactions on Mathematical Software*, 40 (1), 30, 2013.
[22] Gebremedhin, Assefaw Hadish, and Fredrik Manne. "Scalable parallel graph coloring algorithms." *Concurrency: Practice and Experience* 12, no. 12 (2000): 1131-1146.
[23] Grappolo, the Grappolo graph toolkit, https://github.com/luhowardmark/GrappoloTK, last accessed on 5/6/2021.
[24] Grosset, Andre Vincent Pascal, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. “Evaluating graph coloring on GPUs.” *ACM SIGPLAN Notices* 46, no. 8 (2011): 297-298.
[25] Gupta, Kshitij, Jeff A. Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. IEEE, 2012.
[26] Hasenplaugh, William, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. “Ordering heuristics for parallel graph coloring.” In *26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 166-177. ACM, 2014.
[27] Hoffman, Karla L., Manfred Padberg, and Giovanni Rinaldi. "Traveling salesman problem." Encyclopedia of operations research and management science 1 (2013): 1573-1578.
[28] Huang, G., and Weerakorn Ongsakul. “An efficient task allocation algorithm and its use to parallelize irregular Gauss-Seidel type algorithms.” In Proceedings of *8th International Parallel Processing Symposium*, pp. 497-501. IEEE, 1994.
[29] Jaiganesh, Jayadharini, and Martin Burtscher. "A high-performance connected components implementation for GPUs." In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, pp. 92-104. 2018.
[30] Jones, Mark T., and Paul E. Plassmann. “A parallel graph coloring heuristic.” *SIAM Journal on Scientific Computing* 14, no. 3 (1993): 654-669.
[31] Kokkos-Kernels, https://github.com/kokkos/kokkos-kernels, last accessed on 5/6/2021.
[32] Luby, Michael. “A simple parallel algorithm for the maximal independent set problem.” *SIAM journal on computing* 15, no. 4 (1986): 1036-1053.
[33] Martínez-Bazan, Norbert, M. Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. “Efficient graph management based on bitmap indices.” In *16th International Database Engineering & Applications Symposium*, pp. 110-119. ACM, 2012.
[34] Matula, David W., George Marble, and Joel D. Isaacson. “Graph coloring algorithms.” In *Graph theory and computing*, pp. 109-122. Academic Press, 1972.
[35] Merrill, Duane, Michael Garland, and Andrew Grimshaw. “Scalable GPU graph traversal.” In *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117-128. ACM, 2012.
[36] Nasre, Rupesh, Martin Burtscher, and Keshav Pingali. “Data-driven versus topology-driven irregular computations on GPUs.” In *2013 IEEE International Symposium on Parallel and Distributed Processing*, pp. 463-474. IEEE, 2013.
[37] Naumov, Maxim, Patrice Castonguay, and Jonathan Cohen. “Parallel graph coloring with applications to the incomplete-LU factorization on the GPU.” Nvidia White Paper, 2015.
[38] Nilsson, Christian. "Heuristics for the traveling salesman problem." Linkoping University 38 (2003): 00085-9.
[39] Osama, Muhammad, Minh Truong, Carl Yang, Aydın Buluç, and John Owens. "Graph coloring on the GPU." In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 231-240. IEEE, 2019.
[40] Siek, Jeremy, Andrew Lumsdaine, and Lie-Quan Lee. “The boost graph library: user guide and reference manual.” Addison-Wesley, 2002.
[41] Singhal, Nandini, Sathya Peri, and Subrahmanyam Kalyanasundaram. “Practical multi-threaded graph coloring algorithms for shared memory architecture.” In *18th International Conference on Distributed Computing and Networking*, p. 44. ACM, 2017.
[42] SNAP, Stanford Large Network Dataset Collection, https://snap.stanford.edu/data/, last accessed on 5/6/2021.
[43] SuiteSparse Matrix Collection, https://sparse.tamu.edu/, last accessed on 5/6/2021.
[44] Warren, Henry S. Hacker's delight. Pearson Education, 2013.
[45] Welsh, Dominic JA, and Martin B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems.” *The Computer Journal* 10, no. 1 (1967): 85-86.