Increasing the Parallelism of Graph Coloring via Shortcutting

Ghadeer Alabandi
Department of Computer Science
Texas State University
San Marcos, TX, USA
gaa54@txstate.edu

Evan Powers
Department of Computer Science
Texas State University
San Marcos, TX, USA
edp30@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

Abstract

Graph coloring is an assignment of colors to the vertices of a graph such that no two adjacent vertices get the same color. It is a key building block in many applications. Finding a coloring with a minimal number of colors is often only part of the problem. In addition, the solution also needs to be computed quickly. Several parallel implementations exist, but they may suffer from low parallelism depending on the input graph. We present an approach that increases the parallelism without affecting the coloring quality. On 18 test graphs, our technique yields an average of 3.4 times more parallelism. Our CUDA implementation running on a Titan V is 2.9 times faster on average and uses as few or fewer colors as the best GPU codes from the literature.

CCS Concepts

• Computing methodologies → Massively parallel algorithms

Keywords

Graph coloring, shortcuts, 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. The graph coloring problem is the problem of coloring a graph using as few colors as possible. More formally, a vertex coloring of an undirected graph G = (V, E) is a mapping C from vertices to colors such that C(i) ≠ C(j) for every edge (i, j) ∈ E.

Graph coloring is a 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 [6], LU factorization [25], and parallel Gauss-Seidel algorithms for solving non-linear equations [18].

Graph coloring is NP-hard, that is, there is no known polynomial time algorithm that can solve it optimally [13]. However, several heuristic algorithms exist to color a graph using few colors. These algorithms produce a valid coloring, i.e., guarantee that no adjacent vertices have the same color, but they may require more colors than the optimal algorithm, i.e., do not guarantee optimality.

In general, 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 tackling is how to deliver a good coloring quality at high speed. Our solution is to increase the parallelism without loss in quality.

One well-known heuristic is the greedy algorithm. It assigns a random priority to each vertex. Then it repeatedly selects the uncolored vertex that 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 [2][5][17][27] follow the Jones-Plassmann approach [19], i.e., 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.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374519

Several priority heuristics have been proposed for determining the order in which to color the vertices. 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 vertex set, 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 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 [17].

Our algorithm is based on the Jones-Plassmann (JP) approach with the largest-degree-first (LDF) heuristic. We selected JP-LDF as it tends to produce good colorings while being quite fast [17]. 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 minimize this waiting, we have developed "shortcuts" that, under certain conditions, enable us to already color lower-degree vertices before their higher-degree neighbors have been colored. Importantly, these shortcuts are guaranteed to yield the same final coloring as the JP-LDF algorithm without the shortcuts. However, the shortcuts increase the amount of parallelism as more vertices can be colored simultaneously, thus boosting performance.

This paper makes the following main contributions.

- It presents algorithmic optimizations to increase the amount of parallelism in graph coloring without affecting the coloring quality.
- It describes techniques to efficiently implement these algorithmic optimizations.
- It demonstrates that our CUDA implementation is faster than prior CPU and GPU graph coloring codes on a variety of graphs.

The CUDA source code of our implementation 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. Section 3 explains the shortcuts and the optimizations to implement them efficiently. Section 4 summarizes related work. Section 5 describes the methodology. Section 6 presents and analyzes the results. Section 7 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 that is not possible, 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. For simplicity, the vertices are labeled in LDF order: vertices A, B, C, D, and E have degree 5, vertex F has degree 4, and vertex G has degree 3. We use alphabetic ordering to break ties between vertices of the same degree, i.e., letters that appear earlier in the alphabet win the tie. The resulting ordering imposes a direction upon each edge (from the higher-priority vertex that must be colored first to the lower-priority neighbor), which turns the undirected graph into a directed acyclic graph (DAG). The DAG is shown in Figure 1c.

Figure 1d displays a possible 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. 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".

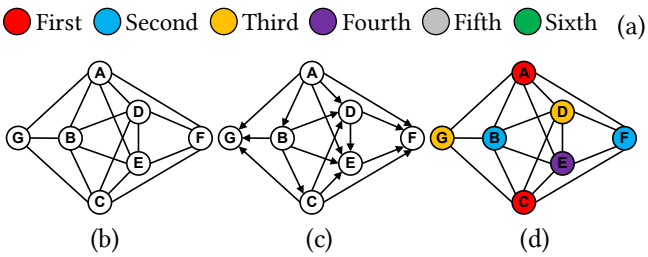

Figure 1: Assumed color order (a), sample graph (b), LDF-imposed DAG (c), and greedy coloring (d)

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 determines the amount of parallelism. Figure 2 illustrates the steps of the JP-LDF algorithm on the sample graph.

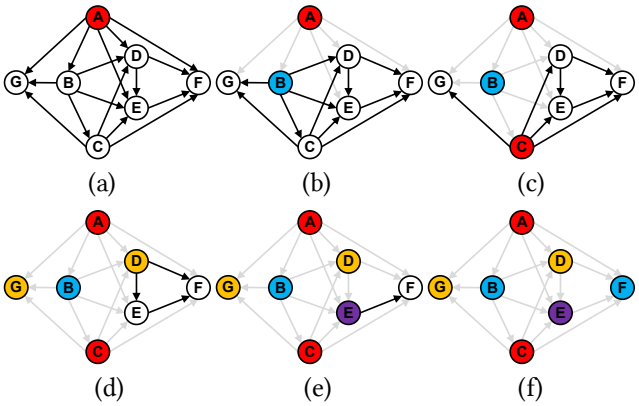

Figure 2: Steps of the JP-LDF algorithm

Figure 2a 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.

In the first processing step (Figure 2b), vertex B has no incoming dark edges. It gets blue as it has a neighbor that already uses red. In the second step (Figure 2c), vertex C has no incoming dark edges, so it is colored with the best available color, which is red. In the third step (Figure 2d), 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 is orange in both cases. In the fourth step (Figure 2e), only vertex E is ready. It must be colored purple as all "better" colors are taken by its neighbors. In the fifth and final processing step (Figure 2f), vertex F is colored blue. Since all vertices are now colored, the JP-LDF algorithm terminates.

3 Shortcut Approach

There is little parallelism in the above example. 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 3a. We reuse the color order from Figure 1a in this section.

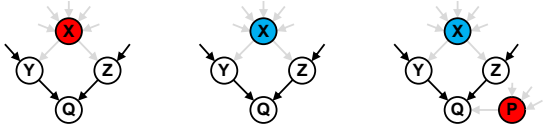

Figure 3: Examples of Shortcut 1

Vertices Y and Z 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 be colored red (the best color) without waiting for Y or Z. This is safe because Y and Z are guaranteed not to use red as they both have a neighbor that is already red. Figure 3b depicts a similar scenario but vertex X is now blue. Applying the same reasoning, we conclude that it is safe to color Q blue as well. However, we want to give each vertex 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 3c, we do not have to wait because blue is the best possible color for Q, and we know that neither Y nor Z 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 record, for each vertex, what colors it is still considering. We call them the "possible colors". This information allows us to decide both the best available color for a vertex and whether a neighbor is still considering a specific color. We store this information in a bitmap, where each bit represents one color. A set bit (1) means the corresponding color is still possible, and a cleared bit (0) means it is not. The position of the bit reflects to which color it refers.

A colored vertex has a single set bit in the bitmap indicating 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 possible.

The bitmaps are initialized with the k+1 bottom-most (least-significant) bits set, where k is the number of incoming DAG edges. This is because, in the worst case, every

incoming edge of vertex v 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 of the 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 [30].

This bitmap-based approach is utilized in other graph-coloring codes, e.g., by Martínez-Bazan et al. [21]. Our approach employs the bitmaps for two additional tasks, namely to determine the best available color of a vertex (the lowest set bit in its bitmap) and whether any of the higher-priority neighbors are still considering this color to determine if the first shortcut can be applied. Moreover, we use the bitmaps for a second type of shortcut.

The second shortcut allows to ignore some higher-priority neighbors before they have been colored, which is tantamount to deleting an edge. This has two benefits. First, it enables us to remove one possible color from the bitmap as 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 that edge. Figure 4 illustrates the idea behind the second shortcut.

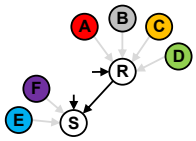

Figure 4: Example of Shortcut 2

In this example, vertex R cannot be colored because it is waiting for one higher-priority neighbor (the incoming dark edge). However, we 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 R and S, no matter which of its possible colors R eventually gets, it will not interfere with S. Hence, we can delete the edge from R to S. That lowers the number of incoming dark edges of vertex S to one, meaning it only needs to consider two possible colors. Consequently, we can safely 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.

It is important to note that neither of the two shortcuts affect the ultimate coloring of the graph in any way. They just speed up the processing by increasing the parallelism.

Figure 5 shows how the sample graph is colored using ECL-GC, our shortcut-based graph-coloring algorithm. In addition to the vertices and edges, the figure includes the bitmap of possible colors for each vertex. Note that the right-most bit represents the first color, the next bit to the left the second color, and so on.

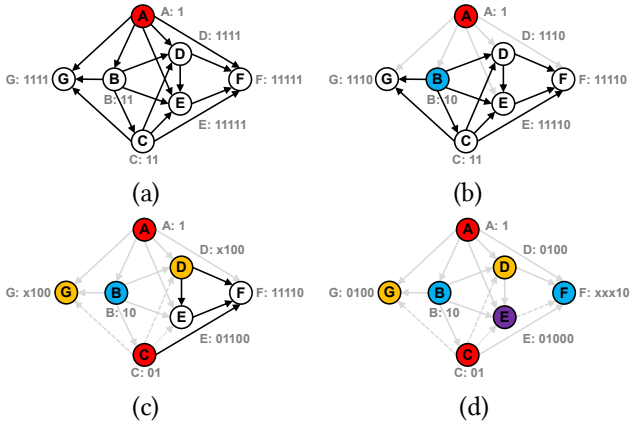

Figure 5: Steps of our ECL-GC algorithm

The initialization phase of ECL-GC (Figure 5a) is identical to that of the JP-LDF algorithm (Figure 2a). Moreover, each vertex gets k+1 bits that are set to one, where k is the number of incoming DAG edges. All non-displayed leading bits are zero. In each of the following computation steps, all uncolored vertices are processed in parallel. Every vertex v visits all incoming edges/neighbors. There are three cases:

1) If the neighbor has been colored, i.e., its bitmap only contains a single set bit, the edge is removed (grayed out) and one bit in the bitmap of v is cleared. If the bit corresponding to the neighbor'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 coloring performed by the JP algorithm. However, if the bit corresponding to the neighbor's color is not set, the highest set bit in the bitmap of v is cleared instead. This is not necessary in the JP algorithm. It is also not required in the ECL-GC algorithm, but it may help with the following two cases.

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

3) For all remaining uncolored 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 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 5b), all vertices that are adjacent to A clear their rightmost 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 second computation step (Figure 5c), multiple events occur. All uncolored vertices that are adjacent to B clear their second 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 bitmap 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 using Shortcut 1 since their best possible color (orange) is not considered by any of their incoming neighbors. Applying Shortcut 1 clears all the 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 (outdated or new) bitmap of C, so the edge CE is deleted and the highest set bit of E is cleared.

In the third computation step (Figure 5d), vertices E and F remove their third 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 using Shortcut 1 and the higher bits are cleared. At this point, all vertices are colored, and the algorithm terminates.

The resulting coloring is identical to that of the serial and JP-LDF algorithms. Moreover, it only takes the ECL-GC algorithm three steps to color this graph compared to the five steps of 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. Furthermore, 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 since there must be at least one such neighbor given that v is uncolored. Assuming vertex n represents one of those neighbors and that set $B(v) \subset C(v)$ contains the best color of $C(v)$, i.e., $|B(v)| = 1$, we end up with the 16 possibilities listed in Table 1.

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	

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 derive a 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 additional shortcuts.

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| colors. We define a graph as sparse if it has O(|V|) edges, that is, $|E| = c|V|$ where c is a small 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 be colored with just a few colors. This observation led us to treat high-degree and low-degree vertices separately. Specifically, we fully implement the shortcuts on the low-degree vertices and only approximate them on the high-degree vertices to avoid the processing of long bitmaps.

For each low-degree vertex ($d(v) < 32$), 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 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. However, the best possible color is maintained 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 simplified implementation runs in constant time (irrespective of how long the bitmaps are) but may miss some shortcutting opportunities. Shortcut 2 is simply skipped as it is less important (cf. Section 6.2.3).

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 produces a deterministic coloring and incorporates the above optimizations. It transfers the graph to the GPU and the final colors back to the CPU. The code repeatedly processes the vertices until convergence is reached. For performance reasons, the processing is done asynchronously, which may result in data races. However, these races are guaranteed to be benign because the bitmaps, once initialized, only ever have bits cleared. Similarly, the first integer (see above) only ever increases. Due to these two types of monotonicity, it is always safe for a thread to act upon an outdated value, but doing so may require extra rounds.

4 Related Work

A large amount of related work exists on graph coloring. However, we know of no other work that proposes the use of 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 [20]. 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) [19] based on Luby's Monte Carlo algorithm. Luby's algorithm selects new random numbers in each iteration, which requires global synchronization (a barrier). 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.

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 [17]. The JP algorithm can easily be augmented with LDF. The operation of the resulting parallel JP-LDF algorithm is outlined in Section 2.

The Smallest-Degree-Last (SDL) algorithm tries to improve upon the coloring quality of LDF by using more sophisticated weights [22]. It comprises two phases, a weighting phase 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 the LDF algorithm 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 [16]. 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 [4].

The CUSPARSE library [8] includes the "csrcolor" graph-coloring code [3]. 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 [5] algorithms. It uses multiple hash functions to generate the "random" numbers for each vertex. The local maximums and minimums of the hash values are used 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. [4] proposed two graph coloring algorithms based on Nasre's ideas for implementing irregular algorithms on GPUs [24]. 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 [4], including bitmap operations to reduce the memory footprint and the time consumed in reading and writing the color mask. For better load balancing, they implemented Merrill's balancing strategy [23], 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.

5 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 producing the least number of colors if it is different.

Table 2. The graph coloring codes we evaluate

Device	Ser/Par	Name	Version	Source
GPU	Parallel	ECL-GC (our code)	1.0	[11]
		CUSP	0.5.1	[9]
		csrcolor	9.2.88	[3]
		Data-wlc	1.0	[4]
		Data-pq	1.0	[4]
CPU	Parallel	GMMP-NT		[7]
		FirstFit	1.0	[4]
		Grappolo		[15]
CPU	Serial	JP-D1		[7]
		FirstFit	1.0	[4]
		Boost	1.66.0	[1]

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, 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.

We present results from two GPUs. The first is a Titan V with 5120 processing elements distributed over 80 multiprocessors. Each multiprocessor has 96 kB of L1 data cache. 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 GeForce GTX Titan X with 3072 processing elements distributed over 24 multiprocessors. Each multiprocessor has 48 kB of L1 data cache. The 24 multiprocessors share a 2 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 336 GB/s.

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_52" for the Titan X. 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≥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) [10], the Galois framework (Galois) [12], the Stanford Network Analysis Platform (SNAP) [28], and the SuiteSparse Matrix Collection (SMC) [29]. 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.

6 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.

6.1 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.

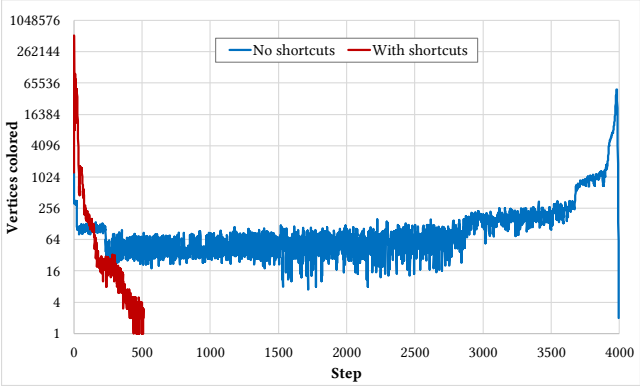

Figure 6: Amount of parallelism in each step on the kron_g500-logn21 graph

Figures 6 and 7 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 6 shows that the shortcuts can yield a large increase in parallelism, in this case a 7.85-fold increase. In contrast, Figure 7 shows the worst case, i.e., an example where the shortcuts do not increase the average parallelism. However, they 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.

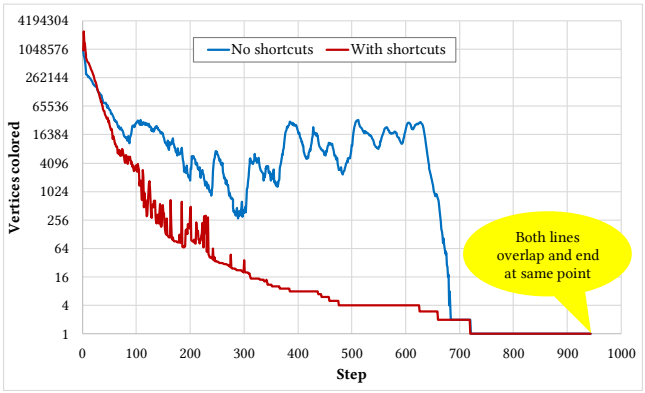

Figure 7: Amount of parallelism in each step on the uk-2002 graph

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 one of the 18 tested graphs. In the best case (rmat22.sym), the parallelism is over 12 times higher. On average, it is 3.4 times higher, demonstrating the potential of the shortcuts.

Figure 8 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 is

identical to 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.

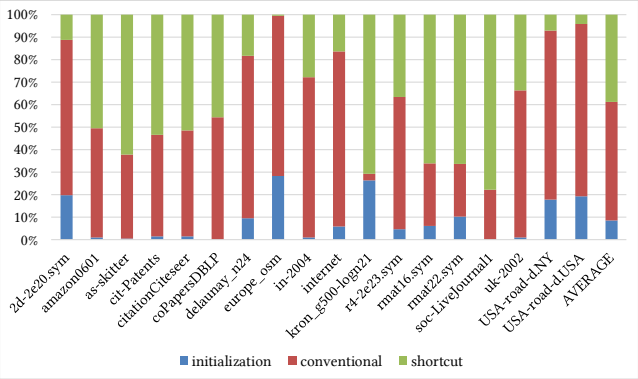

Figure 8: Fraction of colors assigned by various means

6.2 Comparison with GPU Codes

This subsection compares the performance of ECL-GC to that of leading GPU codes. We show results for CUSP and csrcolor as well as Data-wlc and Data-pq, the two fastest versions of Chen et al.'s algorithms described in Section 4.

6.2.1 Coloring Quality

Figure 9 shows the number of colors needed by the five 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.

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 and Data-pq, 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.

ECL-GC either uses the smallest or the same number of colors for all inputs compared to the other four GPU codes. CUSP, Data-wlc, and Data-pq yield a similar coloring quality. Csrcolor requires the largest number of colors on each of the 18 graphs. The geometric mean is 30.6 colors for ECL-GC, 35.0 for CUSP, 149.4 for csrcolor, 37.2 for Data-wlc, and 34.3 colors for Data-pq.

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.

6.2.2 Throughput

Figures 10 and 11 present the throughput of the codes on two GPUs. The x axis lists the inputs and the geometric mean whereas the y axis shows the throughput in millions of completed (colored) vertices per second on a logarithmic scale. Throughput is a higher-is-better metric.

Figure 10 shows the throughput on the Titan V. ECL-GC, which is our implementation with the shortcuts, is faster than CUSP on all tested inputs. It is faster than Data-wlc and Data-pq on 16 of the 18 graphs and faster than csrcolor on all but one input. Based on the geometric mean, ECL-GC is 29.9 times faster than CUSP, 5.5 times faster than csrcolor, 3.7 times faster than Data-wlc, and 2.9 times faster than Data-pq. Note that, in each of the few cases where the other codes are faster, they require more colors.

We correlated the speedup of our code over the other codes with various graph properties and found a moderate linear correlation with both the maximum and the average degree, which is expected because the higher the degree the higher the chance that the algorithm must wait for higher-priority neighbors, which is where the shortcuts can help. In fact, our code outperforms the other codes by at least a factor of two on all tested graphs with a maximum degree

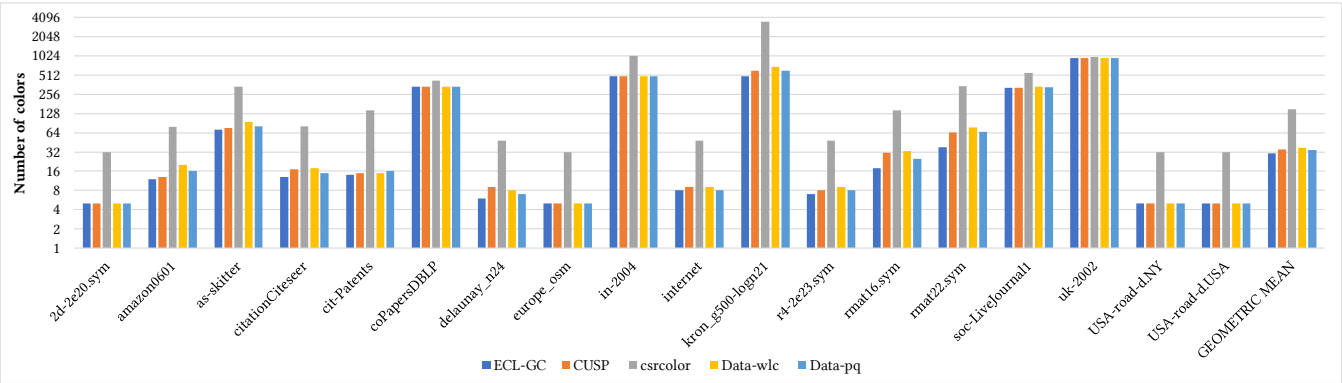

Figure 9: Number of colors needed by the GPU codes

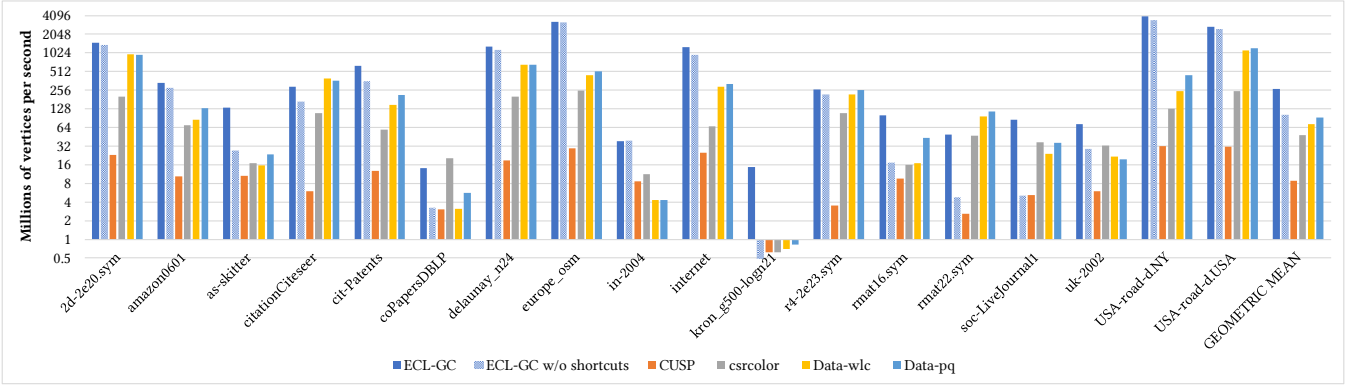

Figure 10: Throughput in millions of completed vertices per second on a Titan V

above 4000. 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 times faster than Data-pq, the second fastest of the five 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 10 also shows results for "ECL-GC w/o shortcuts", which is ECL-GC with the shortcut code disabled. Its geometric-mean performance is slightly higher than that of the other tested codes, meaning our baseline implementation is on par with the best codes from the literature. Section 6.2.3 discusses the performance of the two shortcuts in more detail.

Figure 11 shows throughput results for the older Titan X GPU. ECL-GC outperforms CUSP on all tested inputs. It outperforms Data-wlc and Data-pq on 15 and csrcolor on 17 of the 18 graphs. Again, in all cases where the other codes are faster, they use more colors. Based on the geometric mean, ECL-GC is 12.4 times faster than CUSP, 3.0 times faster than csrcolor, 2.1 times faster than Data-wlc, and 1.9 times faster than Data-pq.

6.2.3 Shortcut Performance

Table 5 presents the performance benefit due to the shortcuts. 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 strongly 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

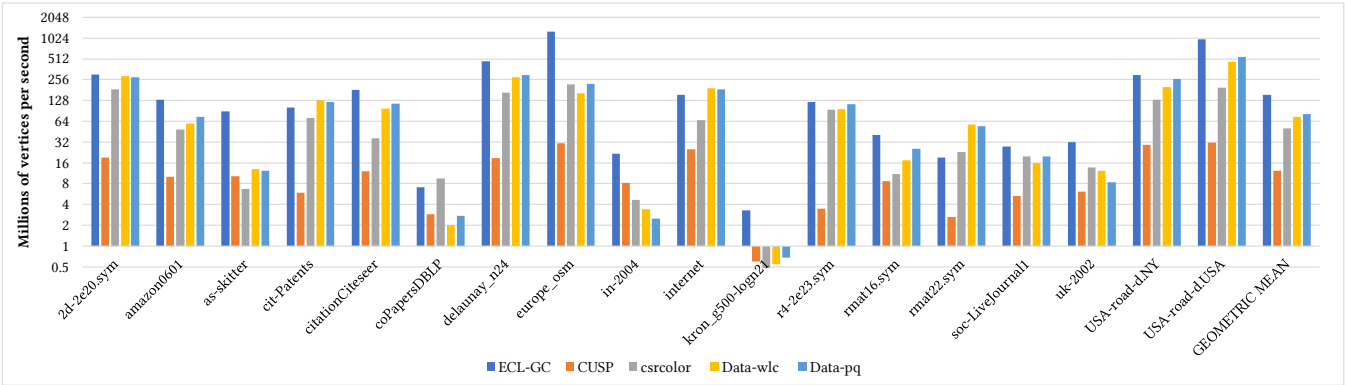

Figure 11: Throughput in millions of completed vertices per second on a Titan X

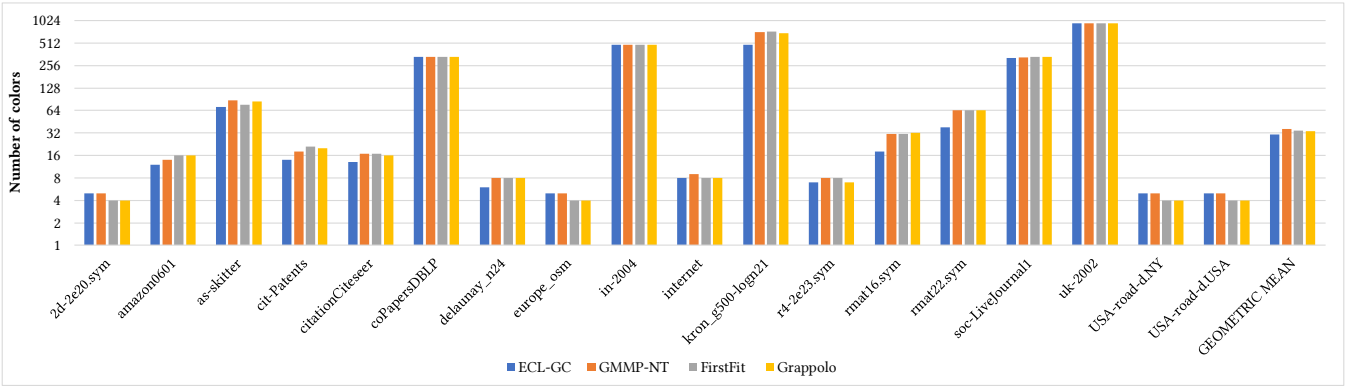

Figure 12: Number of colors needed by the parallel CPU codes as well as by ECL-GC

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

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 where 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.

6.3 Comparison with CPU Codes

In the following subsections, we compare the performance of ECL-GC running on the Titan V to that of the leading parallel and serial CPU codes. Figures 12 and 14 show the number of colors. The x axis lists the inputs and the geometric mean whereas the y axis lists the number of colors using a logarithmic scale. Figures 13 and 15 show the throughput. The x axis again lists the input graphs and the geometric mean whereas the y axis lists the throughput in completed vertices per second on a logarithmic scale.

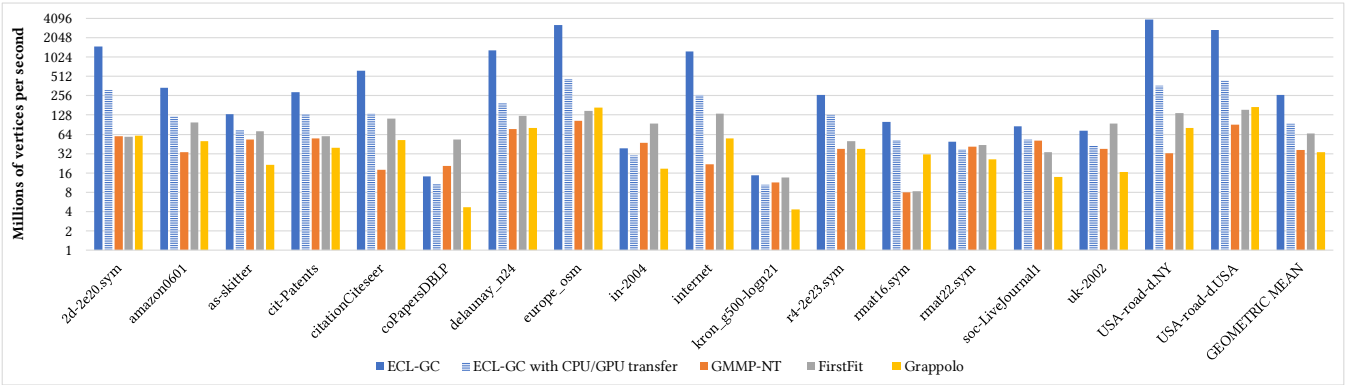

Figure 13: Throughput in millions of completed vertices per second on 20 Xeon cores (Titan V for ECL-GC)

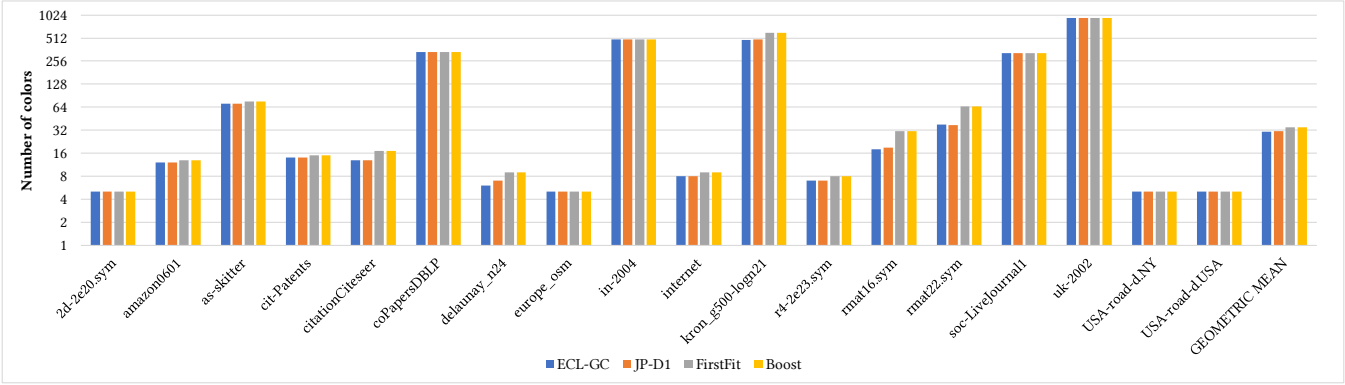

Figure 14: Number of colors needed by the serial CPU codes as well as by ECL-GC

6.3.1 Parallel CPU Performance Comparison

This subsection compares the throughput and coloring quality of ECL-GC to leading parallel CPU codes. We show results for ColPack's GMMP algorithm with the natural (NT) heuristic priority [14], the FirstFit implementation by Chen et al. [4], and the graph-coloring code Grappolo [15].

Figure 12 shows the number of colors assigned by the parallel CPU codes and by ECL-GC. As the number of colors may vary from run to run for GMMP-NT, FirstFit, and Grappolo, we present the minimum number observed. ECL-GC uses fewer colors than ColPack's GMMP-NT on all tested inputs. It uses the smallest or the same number of colors as the FirstFit and Garppolo 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. The geometric mean is 30.6 colors for ECL-GC, 36.0 for GMMP-NT, 34.3 for FirstFit, and 34.0 colors for Grappolo.

Figure 13 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 Figure 13 are for 40 threads since hyperthreading yields a shorter runtime in most cases. ECL-GC running on the Titan V is faster than GMMP-NT and Grappolo on all tested inputs and faster than FirstFit on 15 of the 18 inputs. Based on the geometric mean, ECL-GC is 7.2 times faster than GMMP-NT, 4.0 times faster than FirstFit, and 7.8 times faster than Grappolo on the tested graphs.

For reference, Figure 13 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 a step of a larger computation. If the previous and next steps are also executed on the GPU, no data transfers are needed.

6.3.2 Serial CPU Performance Comparison

This subsection compares the throughput and coloring quality of ECL-GC to leading serial codes. We show results

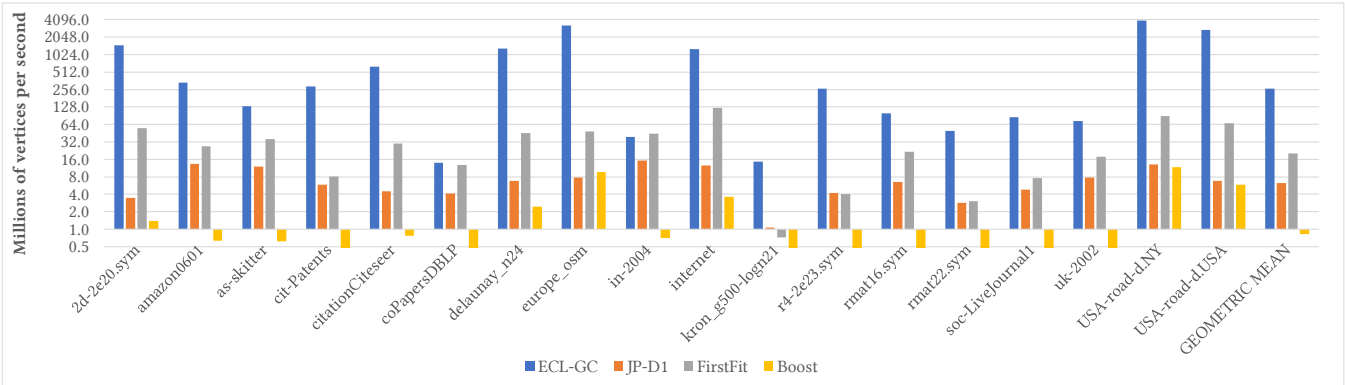

Figure 15: Throughput in millions of completed vertices per second on a Xeon core (Titan V for ECL-GC)

for ColPack's sequential JP code with its fastest heuristic (D1) [7], the serial FirstFit code by Chen et al. [4], and the graph-coloring code in the Boost library [1][26].

Figure 14 presents the number of colors assigned by the serial codes and by ECL-GC. ECL-GC uses fewer or the same number of colors as serial FirstFit and Boost on all tested inputs. ECL-GC's and JP-D1's coloring quality is almost identical. This is not surprising given that JP-D1 and ECL-GC both implement the Jones-Plassmann algorithm with the largest-degree-first heuristic. The small discrepancies on three inputs are due to different tie breakers. The geometric mean is 30.6 colors for ECL-GC, 30.9 colors for ColPack's JP-D1, and 35.0 colors for both FirstFit and Boost.

Figure 15 shows the serial throughput on the Xeon system as well as that of ECL-GC running on the Titan V. ECL-GC is faster on all inputs expect on in-2004. On this graph, on which the shortcuts are nearly ineffective and the average parallelism is low (cf. Table 4), ECL-GC is 16% slower than FirstFit. Based on the geometric mean, ECL-GC is 42.9 times faster than JP-D1, 13.2 times faster than FirstFit, and 324 times faster than Boost.

7 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 3.4 times on average). Under certain conditions, these shortcuts enable the code to 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 for high-degree vertices. The paper also presents optimizations to efficiently implement these shortcuts.

We implemented our approach in CUDA. The code is available at https://cs.txstate.edu/~burtscher/research/ECL-GC/. Running on a Titan V, it is on average 2.9 times faster than the fastest prior GPU code, 4.0 times faster than the fastest OpenMP code running 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.

Comparing the performance across two different GPU generations, we find that our code is 3.1 times faster on the newer GPU whereas the other GPU codes are only up to twice faster. The better scaling of our code to a newer GPU may indicate that it will outperform the other codes by larger margins on future GPUs.

In conclusion, we hope our work will help improve the performance of many applications that incorporate graph coloring as a key step and inspire researchers to develop similar shortcut ideas to increase the amount of parallelism in other important (graph) algorithms.

Acknowledgments

We thank the anonymous reviewers for their feedback, which helped improve our paper. This work was supported in part by the National Science Foundation under award #1406304 and by equipment donations from Nvidia.

References

[1] Boost, https://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/index.html, last accessed on 12/28/2019.

[2] 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.

[3] Chen and Li, https://github.com/chenxuhao/csrcolor, last accessed on 12/28/2019.

[4] 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.

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

[6] 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.

[7] ColPack, Combinatorial Scientific Computing and Petascale Simulations, https://github.com/CSCsw/ColPack, last accessed on 12/28/2019.

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

[9] Dalton, S., and N. Bell. "CUSP: A C++ templated sparse matrix library." http://cusplibrary.github.io, last accessed on 12/28/2019.

[10] DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, http://www.dis.uniroma1.it/challenge9/download.shtml, last accessed on 12/28/2019.

[11] ECL-GC, Texas State University, https://cs.txstate.edu/~burtscher/research/ECL-GC/, last accessed on 12/28/2019.

[12] Galois, ISS - The University of Texas at Austin, https://iss.oden.utexas.edu/?p=projects/galois, last accessed on 12/28/2019.

[13] Garey, Michael R., and David S. Johnson. "Computers and Intractability", vol. 29. W. H. Freeman and Company, New York (2002): 1-99.

[14] 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.

[15] Grappolo, the Grappolo graph toolkit, https://github.com/luhowardmark/GrappoloTK, last accessed on 12/28/2019.

[16] 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.

[17] 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.

[18] 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.

[19] Jones, Mark T., and Paul E. Plassmann. "A parallel graph coloring heuristic." *SIAM Journal on Scientific Computing* 14, no. 3 (1993): 654-669.

[20] Luby, Michael. "A simple parallel algorithm for the maximal independent set problem." *SIAM journal on computing* 15, no. 4 (1986): 1036-1053.

[21] 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.

[22] Matula, David W., George Marble, and Joel D. Isaacson. "Graph coloring algorithms." In *Graph theory and computing*, pp. 109-122. Academic Press, 1972.

[23] Merrill, Duane, Michael Garland, and Andrew Grimshaw. "Scalable GPU graph traversal." In *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117-128. ACM, 2012.

[24] 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.

[25] Naumov, Maxim, Patrice Castonguay, and Jonathan Cohen. "Parallel graph coloring with applications to the incomplete-LU factorization on the GPU." Nvidia White Paper, 2015.

[26] Siek, Jeremy, Andrew Lumsdaine, and Lie-Quan Lee. "The boost graph library: user guide and reference manual." Addison-Wesley, 2002.

[27] 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.

[28] SNAP, Stanford Large Network Dataset Collection, https://snap.stanford.edu/data/, last accessed on 12/28/2019.

[29] SuiteSparse Matrix Collection, https://sparse.tamu.edu/, last accessed on 12/28/2019.

[30] 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.