IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 8 October 2023, accepted 20 October 2023, date of publication 1 November 2023, date of current version 13 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3329082

RESEARCH ARTICLE

An Experimental Evaluation of Graph Coloring Heuristics on Multi- and Many-Core Architectures

ALESSANDRO BORIONE, LORENZO CARDONE, (Graduate Student Member, IEEE), ANDREA CALABRESE, (Member, IEEE), AND STEFANO QUER, (Member, IEEE)
Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Stefano Quer (stefano.quer@polito.it)

ABSTRACT Many modern applications are modeled using graphs of some kind. Given a graph, assigning labels (usually called colors) to vertices is called graph coloring. Colors must be assigned so that no two vertices connected by an edge share the same color. Graph coloring has essential applications in many different fields, and many scalable algorithms have been proposed to solve it efficiently, such that researchers have recently started experimenting with coloring, even on many-core GPU devices. In our work, we selected, analyzed, implemented, and compared state-of-the-art algorithms suited for multi-core CPU and many-core GPU architectures. Our analysis allowed us to discover the advantages and disadvantages of each algorithm, and enabled us to implement new strategies for those algorithms running on CPU and GPU devices. We propose a new technique based on ''value permutation'' and ''index shifting'' that, once applied to the Jones-Plassmann-Luby algorithm can reduce both the runtime and the number of colors. We compare our code on standard graph benchmarks with the two most used state-of-the-art applications, cuSparse's csrColor and Gunrock's implementations, and one innovative approach named Atos. We present extensive results in terms of computation time and quality of the solution. We show that our fastest implementation is able to achieve high average speedups on mesh-like graphs, with a geometric mean (harmonic mean) of 3.16x (3.05x) against Gunrock, 4.09x (3.06x) against cuSparse, and 4.45x (2.21x) against Atos. Nonetheless it proves to be significantly less effective on scale-free graphs, winning consistently only against Gunrock, with geometric mean (harmonic mean) speedups of 2.76x (2.71x) against Gunrock, 0.13x (0.11x) against cuSparse, and 0.03x (0.01x) against Atos. Moreover, it produces 47% fewer colors than cuSparse, 7% fewer colors than Gunrock, and 63% more colors than Atos.

INDEX TERMS Graph, graph algorithms, parallel computing, parallel architectures, parallel applications, algorithm design and analysis.

I. INTRODUCTION

The rapid accumulation of massive graphs from a diversity of disciplines, such as social and biological networks, geographical navigation, Internet routing, databases, and XML indexing, among others, requires fast and scalable graph algorithms.

Graph coloring is one of the many problems applied to graphs that benefit from algorithms that can produce reasonable solutions quickly. Graph coloring is useful in many different fields, such as timetable scheduling [1], [2], register allocation in compiler optimization [3], Sudoku solving [4], parallelization of tasks [5], and many others. Graph coloring aims to assign a label (i.e., a color) to every vertex of the graph, such that adjacent vertices never have the same label. The problem of generating the best solution, i.e., the solution with the least number of labels, is known to be NP-hard [6]. Luckily, many applications that benefit from graph coloring do not strictly require an optimal solution, and a good approximation is often enough. As a consequence, many scalable heuristics [7], [8], [9], [10] have been proposed over the years to approximate the

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Marozzo.

perfect coloring in a reasonable time. However, heuristics typically present a tradeoff between the time to find a solution and the quality of the coloring process such that very often, the fastest heuristics produce the worse results and vice-versa. Moreover, data is being produced and collected faster, encompassing more information than ever, and graphs are also getting larger, causing the need for scalable algorithms to keep up with the times. Thus, researchers are focusing on solving graph coloring on huge graphs, derived from current data, in a faster and more scalable way.

Interestingly, general-purpose computing on GPUs (Graphical Processing Units) is increasingly used to deal with computationally intensive algorithms coming from several domains [11], [13]. The advent of languages such as OpenCL and CUDA has transformed GPUs into highly-parallel systems which scale gracefully, have a considerable bandwidth, and possess enormous computational power. As a consequence, there has been a continuous effort to redesign graph algorithms to exploit GPUs and CUDA, both from NVIDIA, with its NVIDIA Graph Analytic library, and from independent projects, such as the recent Gunrock library.

Our effort in this paper is directed toward developing a fast algorithm to generate a non-trivial coloring.

First of all, we implemented two parallel versions of the Gebremedhin-Manne [7] (GM) algorithm, and a version of the Jones-Plassmann-Luby [8], [9] (JPL) on CPU. The GM approach first colors graphs in parallel, leaving conflicts in the coloring process (i.e., adjacent vertices with the same color), and then it rectifies those inconsistencies with one parallel and one sequential phase. The JPL strategy first finds non-maximal independent sets by assigning random numbers to vertices, and then color each independent set with a single color. We compare the results of our implementations with state-of-the-art approaches, we analyze their inner steps and use them as a starting point for the subsequent phase.

Then, we present a many-core GPU-based design of the coloring algorithm based on the JPL approach. To avoid neighboring nodes assigned with the same random value, we enhance our implementation with a method based on **value permutation** and **index shift**. At the beginning of the coloring process, we generate unique random values by permuting sets of unique items. Then, inspired by the Gunrock implementation, we changed the random values assigned to each node by simulating a new permutation by circularly shifting the array. We analyze the behavior of our value permutation and our index shift strategies on several graphs. On the one hand, we discover that the permutation process may have some overhead on large arrays. Thus, we implement a faster randomization function producing a weaker scattering but with no significant overhead and no penalty for the rest of the process. On the other hand, we show that our shifting strategies could deliver better results when performed with different modules, especially on graphs with linear dependencies on the node list. As a consequence, our final implementation of the value permutation and index shift makes our code faster, allowing a coloring process with fewer colors. Finally, following the Cohen-Castonguay and the Gunrock implementation, we also find and color two independent sets of nodes per iteration [14], one of the local maxima and one of the local minima.

We compare our versions against two state-of-the-art GPU implementations, NVIDIA's cuSparse library [10] and the Gunrock framework [15], and against a task-based approach for solving graph-related problems, i.e., Atos [16]. We present the number of colors used and the time required to perform the pre-processing, coloring, and post-processing phases on publicly available benchmarks. When we concentrate on the core (coloring) phase, we illustrate that our fastest implementation presents geomean (harmean) speedups of 3.16x (3.05x) against Gunrock, 4.09x (3.06x) against cuSparse, and 4.45x (2.21x) against Atos on graphs with low average degree. Nonetheless it is slower on scale-free graphs and ones with high average degree, presenting geomean (harmean) speedups of 2.76x (2.71x) against Gunrock, 0.13x (0.11x) against cuSparse, and 0.03x (0.01x) against Atos. When we concentrate on the entire process (pre-processing, processing, and post-processing phases, including transfer times), we present geomean (peak) speedups of 7.43x (61.07x) against Gunrock. Moreover, our fastest technique produces 47% fewer colors than cuSparse and 7% fewer colors than Gunrock, based on the same JPL approach. Furthermore, it generates 63% more colors than Atos based on the GM approach, which is known to be slower but produces better coloring results.

Our code and all related experiments are currently available in an open-source repository at https://github.com/stefanoquer/graph-coloring-code and are free for use by external developers.

A. ROADMAP

We organized the paper as follows. Section II introduces our notation, the related works on graph coloring, and the necessary background on SIMT (Single Instruction, Multiple Thread) architectures. Section III presents our implementations for several state-of-the-art algorithms, identifying their advantages and disadvantages. Our experience in the area allowed us to write the procedure presented in Section IV. Section V discusses our experimental analysis, comparing our implementation as presented in Section IV with the fastest versions described in Section III. Section VI concludes the paper by summarizing our work and reporting some comments on possible future works.

II. BACKGROUND

A. NOTATION

We refer to a graph G as $G = (V, E)$, where $V = \{v_1, v_2, \ldots v_n\}$ is the set of vertices, and $E \subseteq V \times V$ the set of edges. More specifically, we will manipulate undirected graphs, where an edge $e \in E$ is an unsorted pair of vertices

$$(v, u) \in E \iff (u, v) \in E \quad \forall v, u \in V$$

We also refer to the cardinality of V and E with n and m, respectively. Moreover, we use the notation

$$adj(v) = \{u \mid (u, v) \in E\}$$

to indicate the adjacency list of v, i.e., the set of nodes that share an edge with v.

Given a graph G, an independent set of nodes I is defined as:

$$I = \{v, u \mid (v, u) \notin E, \forall v, u \in V, v \neq u\}$$

that is, the independent set I includes vertices that are not adjacent. A maximal independent set is an independent set that is not a subset of a larger independent set.

In our implementations, we store a graph G adopting the so-called *Compressed Sparse Row* (CSR) representation. CSR is particularly efficient when large and sparse graphs must be represented since it is a matrix-based representation that stores only non-zero elements of every row. Using this strategy, we can offer fast access to the information related to each row, avoiding useless overhead for very sparse matrices at the same time. Essentially, in the CSR format, edges are represented as a concatenation of all adjacency lists of every node. One additional array is used to index the adjacency list of each vertex in the main array. It is also possible to use a third array to store information about the weight of each edge. We will not use this additional array since we work with non-weighted graphs. Figure 1b reports the CSR representation for the graph of Figure 1a. As an example, in order to iterate through the neighbors of node 1, we would have to access the elements of the adjacency list array starting from the node at position 3 and ending at position 4, since the next node would start at position 5. In general, $adj(v)$ is the sub-array of the adjacency list array starting from $indexes[v]$ included, and ending at $indexes[v+1]$ excluded.

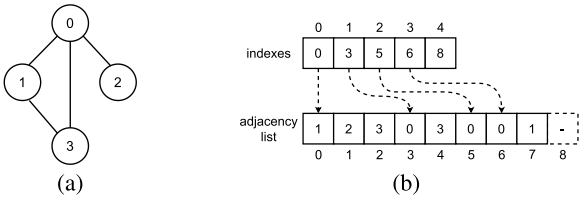


FIGURE 1. A small example of the Compressed Sparse Row (CSR) format: Graph (a) and corresponding CSR representation (b).

B. GRAPH COLORING

Given a graph G, the target of graph coloring is to assign a color $color(v)$ to every vertex $v \in V$, such that if $u \in adj(v)$ then $color(v) \neq color(u)$. It is worth noting that the graph coloring problem is well-defined only for undirected graphs; given that no pair of adjacent vertices can have the same color, it is required that the property of being adjacent is symmetric. In other words, if vertex v is adjacent to vertex u, vertex u must also be adjacent to vertex v. For this reason, we run our experiments on graphs that are either undirected or are directed but have been pre-processed to double all their edges.

The classical approach to graph coloring sequentially visits all vertices $v \in V$, and assigns to each of them the color identified by the lowest number not yet assigned to its neighbors. Algorithm 1 reports the pseudo-code of this greedy approach. The quality of the solution depends on the order in which the nodes are considered. There exists a specific ordering that generates the optimal solution with the least number of colors possible, but finding this ordering is NP-hard [6]. Different heuristics have been proposed as approximate orderings. For example, the Largest-Degree First (LDF) heuristic [2], which colors vertices in order of decreasing degree, usually produces surprisingly good results. Unfortunately, albeit very simple, the greedy algorithm is inherently sequential and difficult to parallelize without major modifications.

Algorithm 1 Greedy Graph Coloring.

Greedy $(G = (V, E), colors)$
1: $V' \leftarrow V$
2: **for** $i = 1$ *to* n **do**
3: Choose a vertex v_i from V'
4: $color(v_i) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v_i)\}$
5: $V' \leftarrow V' \setminus \{v_i\}$
6: **end for**

C. GPUS AND CUDA

With the current technology, GPUs (Graphical Processing Units) can be used well outside the area of computer graphics to solve general-purpose problems. To enable GPGPU (General Purpose GPU) programming, we use a GPGPU software framework to exchange data bi-directionally between the CPU and GPU memory, and we program the GPU's many threads to solve the problem on that data concurrently. Two main frameworks lead the market: OpenCL is the primary open-source framework, and CUDA is the leading proprietary one. In general, if a platform supports both OpenCL and CUDA, the latter is preferable, as the proprietary integration is always excellent, thus leading to a slight increase in execution speed. The algorithms presented in this paper rely on the CUDA framework for their implementation.

The CUDA framework makes use of streaming processors to run tasks in parallel. Each streaming processor in a CUDA-enabled GPU manages threads running on CUDA cores in a way that cores of the same streaming multiprocessor execute the same instruction simultaneously. This implementation of the SIMD (Single Instruction Multiple Data) paradigm is called SIMT (Single Instruction Multiple Thread). Threads are usually scheduled as groups of 32 elements, called "warps". Threads can only access the GPU-dedicated memory, so data must be transferred with the appropriate framework API. Memory access pattern from threads is significant in GPGPU, and can greatly undermine an application's performance. CUDA receives memory requests from the CUDA cores, and joins the

ones referring to adjacent portions of memory to lower the long latency of memory operations. Some applications, especially graph applications like the one we study, have an unpredictable access pattern, making the memory throughput a bottleneck during the execution. However, CUDA also provides the CUDA Stream[1] data structure. A stream is a sequential set of operations, such as memory allocations, memory transfers, and kernel calls, that need to be performed on the GPU. Different streams can perform different operations concurrently on the same GPU, and an implicit synchronization is performed after each operation. Synchronization with the CPU must be explicit, thus allowing the program to perform more operations sequentially on GPU without the CPU waiting for the results.

In general, not all problems benefit from being executed on a GPU. GPUs typically work on matrices representing pixels on the screen, applying mathematical transformations to those pixels following the fast refresh rates of the screen. Because of this, the ideal problems that a programmer can solve via GPGPU are those that present a wide degree of data parallelism, where the operations to be performed are independent of each other and can be computed in parallel.

III. COLORING ALGORITHM

We develop two versions of the Gebremedhin-Manne and one of Jones-Plassmann-Luby algorithm for multi-core CPUs. We also present two versions of the Jones-Plassmann-Luby approach for many-core GPU architectures.

We compare our versions, architecturally and experimentally, with the csrColor routine from the cuSparse library [17], the graph coloring program distributed with the Gunrock library [18], and the task-based approach implemented in Atos [16].

We analyze these algorithms in the following sections.

A. GEBREMEDHIN-MANNE

Gebremedhin and Manne [7] propose a parallel graph coloring algorithm whose core idea is to allow inconsistencies in the coloring process. Specifically, they first divide the vertices into p blocks. Then, they mock-color the vertices of all blocks in parallel. We use the term "mock-color" because the resulting coloring may present a conflict every time two (or more) adjacent vertices are colored by two (or more) different threads simultaneously. Thus, the mock-coloring phase is followed by a parallel phase to discover all conflicts and a final sequential phase where the conflicts are rectified. They also present an improved version of the same algorithm to reduce the number of colors generated during the mock-coloring step.

Algorithm 2 and Algorithm 3 show the standard and an improved algorithms, respectively.

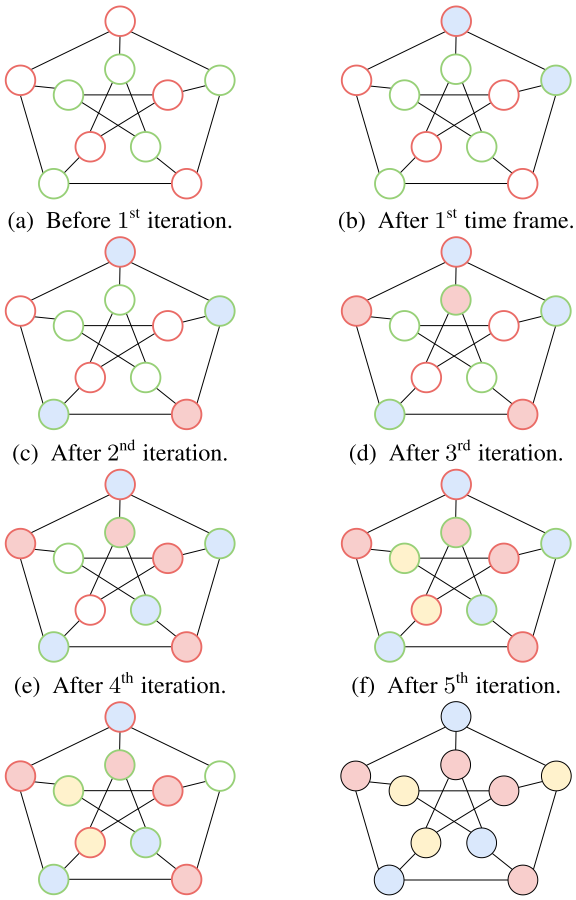
Algorithm 2 is formed by three sections, each corresponding to one for-loop. In the first part (line 1), every graph node is colored in parallel, allowing for coloring errors, i.e., two adjacent nodes can be assigned the same color. In the second part (line 6), the errors generated in before are found. Each pair of neighboring nodes is checked in parallel so that conflicting pairs are saved to be managed later. The number of pairs to check can be reduced by only considering pairs of nodes that were colored at the same time frame during the first part of the process. Time frames, or steps, are a consequence of using a barrier in the first part. As only the nodes that are colored at the same time can present a conflict in the colors assigned, and race conditions between the working threads cause these conflicts, the authors suggest to reduce the pairs of neighbors to avoid coloring inconsistencies. In Algorithm 2, this is shown in the set of nodes S colored in the same step as the current node on line 7, and on the intersection $adj(v) \cap S$ on line 8. Thus, K represents the nodes that need to be recolored. In the third part (line 14), the nodes that were in conflict are recolored, this time sequentially, to avoid adding more coloring errors that would need to be discovered and corrected in the same way. The standard algorithm is relatively slow, as the slowest thread, i.e., the one that colors the node with the most neighbors each time frame, blocks the other threads on the barrier synchronization on line 3. Experimentally, this is shown to take up between 80% and 90% of the execution time.

Algorithm 2 Gebremedhin-Manne Standard Algorithm.

Gebremedhin-Manne-standard $(G = (V, E), colors)$
1: **for** $v \in V$ in parallel **do**
2: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
3: Barrier wait
4: **end for**
5: $K \leftarrow \emptyset$
6: **for** $v \in V$ in parallel **do**
7: $S \leftarrow$ nodes colored in the same step as v in line 2
8: **for** $u \in (adj(v) \cap S)$ **do**
9: **if** $colors(v) = colors(u)$ **then**
10: $K \leftarrow K \cup \min\{v, u\}$
11: **end if**
12: **end for**
13: **end for**
14: **for** $v \in K$ **do**
15: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
16: **end for**

Figure 2 shows a simple graph being colored following Gebremedhin-Manne standard formulation using two blocks. Nodes with the same border color (red or green) belong to the same block. We also assume that processors color the nodes based on the status of the previous time frame, without entering race conditions within the same frame. This assumption is a simplification, as the actual behavior depends on how the processors are scheduled at runtime. Nodes within a block are colored in lexicographical order, which we assume is clockwise, outer to inner, starting from the top-most node. We report the state of the coloring after each time frame in

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams



(a) Before 1st iteration. (b) After 1st time frame.
(c) After 2nd iteration. (d) After 3rd iteration.
(e) After 4th iteration. (f) After 5th iteration.
(g) After the conflict search. (h) After the conflict resolution.

FIGURE 2. Application of the Standard Gebremedhin-Manne algorithm on a small graph of 10 nodes.

Figures 2b, 2c, 2d, 2e, and 2f. Figure 2g shows the state after the conflict search step, which only detects a single conflict. Lastly, Figure 2h reports the final coloring after the conflict correction step is completed. The solution produced uses three colors, which we know is the lower bound to the number of colors for this graph.

Algorithm 3 tries to reduce the number of colors produced by Algorithm 2. It can be logically divided into four parts. Parts one, three, and four are equivalent to the ones appearing in the standard algorithm, in parts one, two, and three, respectively. Part two performs a second coloring, still allowing for mistakes. The coloring is performed by first dividing the nodes into color classes. A color class is the set of nodes assigned the same color in part one of the algorithm. Then the coloring from part one is deleted, and all nodes are recolored in parallel, starting from the ones belonging to the color class that was assigned the largest color in part one, and continuing with the color classes assigned with lower colors, until all nodes are colored a second time. At the end of part four, the improved algorithm finds a solution using a number of colors lower or equal to the number of colors used during the first coloring at the end of part one [7]. Despite producing

Algorithm 3 Gebremedhin-Manne Improved Algorithm

Gebremedhin-Manne-improved $(G = (V, E), colors)$
1: $colors' \leftarrow colors$
2: **for** $v \in V$ in parallel **do**
3: $colors'(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors'(u) \mid u \in adj(v)\}$
4: Barrier wait
5: **end for**
6: **for** k from $\max_{v \in V} colors'(v)$ down to $\min_{v \in V} colors'(v)$ **do**
7: $ColorClass \leftarrow \{v \in V \mid colors'(v) = k\}$
8: **for** $v \in ColorClass$ in parallel **do**
9: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
10: **end for**
11: Barrier wait
12: **end for**
13: $K \leftarrow \emptyset$
14: **for** $v \in V$ in parallel **do**
15: **for** $u \in adj(v)$ **do**
16: **if** $colors(v) = colors(u)$ **then**
17: $K \leftarrow K \cup \min\{v, u\}$
18: **end if**
19: **end for**
20: **end for**
21: **for** $v \in K$ **do**
22: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
23: **end for**

better coloring, the improved algorithm is slower than the standard algorithm, as the coloring is performed twice. The conflicts after part two are fewer than with the standard algorithm, as shown empirically in the original research.

The two previous procedures can be considered as "synchronous" as all threads wait on a barrier (Algorithm 2 on line 3, and Algorithm 3 on lines 4 and 11) before proceeding to the next iteration. Unfortunately, in these algorithms, 90% of the time is spent waiting on the barriers. Thus, Algorithm 2 can be converted to an asynchronous version by removing the barrier wait synchronization on line 3; consequently, line 7 is substituted with $S \leftarrow V$ as, without the barrier, there are no more time frames. Similarly, Algorithm 3 can be made asynchronous by removing the barrier wait call on lines 4 and 11. The asynchronous formulations run faster than their synchronous counterpart as the bottleneck on the barrier synchronization is removed. However, the coloring produced by the asynchronous algorithm is more affected by the execution schedule and generally uses more colors than the synchronous algorithm.

In their paper, Gebremedhin and Manne propose to divide the n nodes to be colored in p blocks of n/p nodes. Each block V_i is then assigned to a processor p_i, with $(1 \leq i \leq p)$, that works on that block alone. The paper fails to address how the nodes are distributed across the blocks. The chosen distribution rule determines how, in cases where (n/p) is not an integer, the remaining $(n \bmod p)$ nodes are distributed into the partitions. In our implementation of the algorithms, we decided to assign the nodes to the blocks based on their

lexicographical ordering. The first p nodes are assigned one to each block, then the second p nodes, and so on. Each block V_i is composed of the nodes:

$$\{v_{k*p+i} \mid 1 \leq k \leq \lceil n/p \rceil\}$$

like in the example of Figure 2. With our distribution policy, if (n/p) is not an integer, we consider $(n \bmod p)$ more nodes that are not part of the original set V. We call these "ghost" nodes, and they are distributed so that all blocks are of the same size $\lceil n/p \rceil$. Ghost nodes are dummy nodes and do not belong to the original graph, so they do not need to be colored. Instead, in the synchronous versions of the algorithm, they serve the purpose of keeping all processors inside the first for-loop (Algorithm 2 line 1 and Algorithm 3 line 2). In this way, all processors leave the for-loop after $\lceil n/p \rceil$ iterations, and there is no need to resize the barrier to accommodate processors that would exit earlier. For the sake of brevity and simplicity, we did not add the block-separator steps in Algorithm 3, since the algorithm itself still implies that the concurrency is limited by the number of processors.

B. ATOS

Chen et al. [16] propose Atos, i.e., a parallel task-based methodology applicable to GPUs that derives from the GM approach. Atos proposes to run a pair of GPU kernels split over two different algorithms. The first kernel works on a frontier consisting of the non-colored nodes and assigns a color to it in a Gebremedhin-Manne-like manner. In practice, the kernel selects a color based on the node's neighborhood, and then adds the node to the frontier of the second kernel. The second kernel checks the correctness of the assignment, making sure that two adjacent nodes never have the same color. It then consumes the nodes newly colored by the first kernel and treats them differently based on the success or failure of the check. In the case of no conflict, the node is removed from the frontier and is permanently colored; on the contrary, in the case of a conflict the node is reassigned to the frontier of the first kernel, forcing it to a new coloring phase. The process continues until all nodes have been permanently colored and no conflict is present anymore. The approach has been designed to prevent the two main problems of Bulk Synchronous Parallel algorithms, i.e., programs that fully utilize the GPU launching a single GPU-wide kernel. The two problems are Load Imbalance and Small Frontier. The former of the two occurs when some threads are faced with significantly more computations than others. The latter happens when there are fewer processes available than the number of threads available.

C. JONES-PLASSMANN-LUBY

Luby [8] suggests that an independent set of nodes can be colored in parallel with the same color without conflicts, and develops an algorithm to find maximal independent sets in a graph.

Jones and Plassmann [9] develop Luby's approach using independent sets. Their strategy finds non-maximal independent sets by assigning a random number to each vertex, and selecting all nodes whose random numbers are local maxima. Each node in the independent set is then colored separately in parallel with the lowest color not assigned to one of their neighbors.

We define the Jones-Plassmann-Luby procedure (JPL) as a middle ground between Jones-Plassmann and Luby's algorithm. First, we find non-maximal independent sets using the random values approach from the Jones-Plassmann algorithm. Then, we color each independent set with a single color like Luby's algorithm. The process is repeated until all nodes are colored. The JPL procedure is reported in Algorithm 4. First, every node v is assigned a random number $\rho(v)$. Then, the algorithm iterates until all nodes are colored. At each step, i of the iteration, an independent set I_i is computed using random numbers. More specifically, a node v is part of I_i if and only if it is yet to be colored, and every one of its neighbors $u \in adj(v)$ is assigned a random value $\rho(u)$ so that $\rho(v) > \rho(u)$. We can say that, if $v \in I_i$, v is a local maximum because its assigned random value is a maximum in the locality of its (non-colored) neighbors. The iteration ends when all members of I_i are assigned with the color i.

Algorithm 4 Jones-Plassmann-Luby Coloring Heuristic

```
JPL-color (G = (V, E))
 1: N ← V
 2: i ← 1
 3: for v ∈ N do
 4:     ρ(v) ← random number
 5: end for
 6: while N ≠ ∅ do
 7:     I ← ∅
 8:     for v ∈ N in parallel on GPU do
 9:         I ← I ∪ {v}
10:         for u ∈ (adj(v) ∩ N) do
11:             if ρ(v) ≤ ρ(u) then
12:                 I ← I \ {v}
13:             end if
14:         end for
15:     end for
16:     for v ∈ I in parallel do
17:         color(v) ← i
18:     end for
19:     N ← N \ I
20:     i ← i + 1
21: end while
```

Following this logic, Algorithm 4 is divided into two main loops. The first iteration (starting at line 3) contains the initialization of the random values associated with each node in the graph. The second cycle colors the graph, and it is divided into two more sections, i.e., the computation of an independent set (lines 8–15), and the actual coloring (line 17). The introduction of independent sets allows

coloring nodes in parallel without risking inconsistencies in coloring adjacent nodes. However, this procedure quickly computes each independent set, but then colors each one of them independently; thus, it may require a large number of iterations. This situation presents itself when sets of nodes with increasing random identifiers are adjacent and form a linear sequence. For example, let us suppose that a node with label 1 is adjacent to a node with label 2, which, in turn, is adjacent to a node with label 3. As a consequence, only one node per iteration will be colored, as node 3 will be colored first, followed by node 2 during the second iteration, and node 1 during the last loop.

Figure 3 shows the JPL algorithm applied to an example graph. Each random number $\rho(v)$ is displayed inside the corresponding node v that is not colored. For the sake of simplicity, we use random integer numbers between 0 and 99. Figures 3b, 3c, 3d, 3e, and 3f show the state of the coloring after each iteration. In Figure 3a, when all nodes are yet to be colored, we can notice the linear sequence of nodes with random numbers $78 \rightarrow 62 \rightarrow 57 \rightarrow 40 \rightarrow 13$, and how the length of the chain (5) is equal to the number of iterations required to complete the coloring.

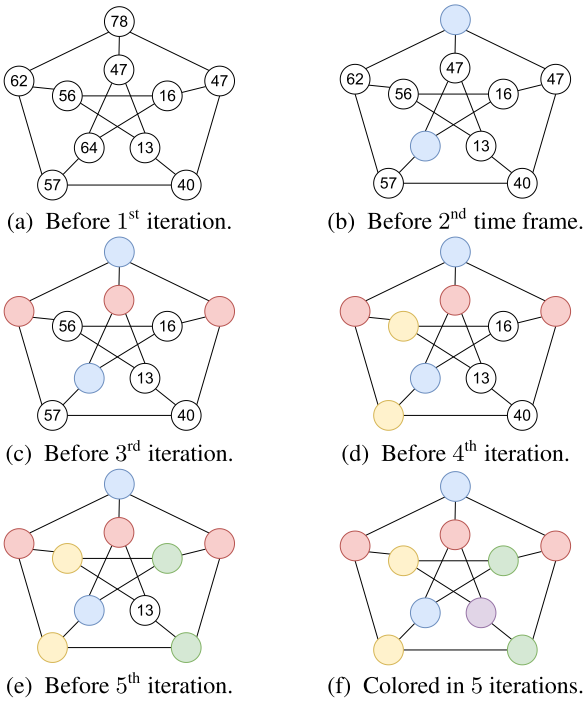


FIGURE 3. Application of the Jones-Plassmann-Luby algorithm on a small graph of 10 nodes.

D. COHEN-CASTONGUAY

Cohen and Castonguay [10] present a GPU-based algorithm for graph coloring derived from the JPL algorithm. They suggest three critical modifications to the original algorithm.

The first suggestion consists in improving the number of nodes that can be concurrently colored at each iteration. To maintain the efficiency of the original algorithm, they propose a way to select two independent sets that are disjointed with little or no extra complexity. In each iteration, the authors search the set of local maxima I_i^M, as shown in Section III-C, and the set of local minima I_i^m. These two sets are independent, which means that there are not adjacent node pairs belonging to the same set. This property holds because it is not possible for two adjacent nodes to simultaneously share the property of having the highest (for the set I_i^M) and the lowest weight (for the set I_i^m) weight among all their neighbors. Moreover, except for nodes with no neighbors, the two sets are also disjoint since it is not possible for a node with neighbors to possess a weight simultaneously larger and smaller than the weights of every other adjacent node. These two properties allow us to color the two sets in parallel with two distinct colors in a single pass [14]. Furthermore, they prove that a parallel algorithm cannot select more than two disjoint independent sets per iteration in the JPL function [14].

The second modification is based on the observation that the vector of random values resides in memory, and each access to one of its element is inherently slow. They suggest that it is possible to disregard the necessity to have the vector of random values by using the hash function $H{:}V \rightarrow K$, where K is a general set where its members can be ordered. A hash function can compute seemingly random values if given the node identifiers as input. The results can be maintained in registers for fast access, then discarded, and recomputed on the fly when needed again. Even if each number must be recomputed several times, the strategy is faster than accessing the value in the main memory.

The third modification is based on the idea of using k hash functions $H_1, \ldots H_k$, thus finding more than two sets per iteration. It must be noted that a function H_i generates two disjoint independent sets, but the two sets, generally, are not disjoint from the two sets generated by another function H_j. Thus the hash functions must be sorted, and a lower-ranking function can consider only nodes not colored by the higher-ranking functions. By using k hash functions, it is possible to color $2k$ sets at a time, significantly reducing the number of iterations needed to color the whole graph. However, the number of hash functions k must be carefully chosen. Having too many hash functions may reduce the speed of the algorithm and hinder the quality of the solution as overlapping independent sets need to be made disjoint before coloring, adding overhead to the algorithm.

These observations make the algorithm perfect for SIMT architecture, where the main bottlenecks are often the memory bandwidth and global synchronization needed to run an algorithm like JPL. Since the hash functions implemented in the algorithm do not change, Cohen-Castonguay is a deterministic algorithm, meaning that given the same input, the output will always be the same as well. Among the algorithms that we considered, this is the only one to have this property intrinsically implemented. The Jones-Plassmann-Luby algorithm can also be adapted to ensure a deterministic result by using a seed for the random number

generation. Similarly, Gunrock and our implementation, based on JPL, can be modified accordingly. The Cohen-Castonguay algorithm is made available through the csrColor routine of the cuSparse library [17].

E. GUNROCK

Gunrock [18] is an open-source library designed to solve graph processing problems on CUDA-enabled GPUs. Gunrock is distributed with a wide variety of graph primitives, among which there is a graph coloring primitive. Gunrock's graph coloring algorithm follows the research by Osama et al. [15]; the implementation, described in Algorithm 5, follows a variation of the JPL algorithm. Similar to the Cohen-Castonguay algorithm presented in Section III-D, Gunrock searches for two independent sets per iteration. However, it retains the vector of random values, called *rand* in Algorithm 5. Moreover, their algorithm does not need any form of load balancing that would imply conspicuous time overheads. Their tests show that their implementation is the fastest one on GPUs.

Algorithm 5 Gunrock Coloring Procedure.

```
Gunrock-color (G = (V, E), rand, colors)
 1: i ← 0
 2: N ← V
 3: while N ≠ ∅ do
 4:     if i mod 2 = 0 then
 5:         for v ∈ N in parallel on GPU do
 6:             rand(v) ← random number
 7:         end for
 8:     end if
 9:     c ← 2 * i + 1
10:     k ← 2 * i + 2
11:     for v ∈ N in parallel on GPU do
12:         I^M ← I^M ∪ {v}
13:         I^m ← I^m ∪ {v}
14:         for u ∈ (adj(v) ∩ N) do
15:             if rand(v) ≤ rand(u) then
16:                 I^M ← I^M \ {v}
17:             end if
18:             if rand(v) ≥ rand(u) then
19:                 I^m ← I^m \ {v}
20:             end if
21:         end for
22:     end for
23:     for v ∈ I^M in parallel on GPU do
24:         colors(v) ← c
25:     end for
26:     for v ∈ (I^m \ I^M) in parallel on GPU do
27:         colors(v) ← k
28:     end for
29:     N ← N \ I^M
30:     N ← N \ I^m
31:     i ← i + 1
32: end while
```

In our experiments, we adopted the Gunrock library within the development branch dated 15 November 2021. This version of the coloring algorithm has a flaw [19], causing infinite loops and preventing the program from finishing with a correct solution. The problem is caused by the management of the *rand* vector. Initially, the vector is populated with random, single-precision floating point values. In lines 15 and 18 of Algorithm 5, the process compares two values of the vector to choose the node v to remove from the corresponding independent set. However, if $rand(v) = rand(u)$, nodes v and u are both removed from both sets by their respective threads. As a consequence, both v and u are never part of an independent set, meaning that they will never be colored, thus leading to an infinite loop as the algorithm terminates when all nodes are colored.

We propose two different approaches to fix this problem.

Our first approach follows the observation that a simple tie-breaking condition would solve the issue when comparing the two values. Thus, we substitute the conditional on line 15 with $rand(v) < rand(u)$ *or* $(rand(v) = rand(u)$ & $v < u)$ and the conditional on line 18 with $rand(v) > rand(u)$ *or* $(rand(v) = rand(u)$ & $v > u)$. In this new version, the tie is broken with a comparison of the two node indexes, which are unique by definition. In other words, a node v is considered a local maximum if $rand(v) \geq rand(w)$ & $v > u, \forall w \in N, \forall u \in M$, where $N = \{w_1, \dots w_k\}$ is the set of the non-colored nodes adjacent to v, and $M = \{u_1 \dots u_b\} \subseteq N$ is the subset of N where $rand(v) = rand(u_i)$. Similarly, a node v is considered a local minimum if $rand(v) \leq rand(w)$ & $v < u$, $\forall w \in N, \forall u \in M$.

Our other approach takes inspiration from previous versions of the coloring primitives where the *rand* vector had its values regenerated every other iteration. This approach fixes the issue because if v and u are assigned the random values $rand_i(v) = rand_i(u)$ at iteration i, it is expected that at iteration $j \geq i + 2$, $rand_j(v) \neq rand_j(u)$. In this way, two adjacent nodes can share the random maximum or minimum at some point, but will be colored at a later iteration, when the random values are eventually different. We get confirmation from the Gunrock developers that this second approach enables the intended behavior of their tool [19]. In Section V, we refer to the Gunrock implementation complete with this approved fix. The regeneration happens in the block on line 4 of Algorithm 5.

In Figure 4, we report an example of a graph colored with the JPL implementation from Gunrock. The graph used and the associated random numbers are the same as adopted in Figure 4, to simplify the comparison of the two algorithms. Figures 4b, 4c, and 4d show the state of the graph after each iteration of the algorithm. The chain of nodes with random numbers $78 \rightarrow 62 \rightarrow 57 \rightarrow 40 \rightarrow 13$ is colored from both ends simultaneously, thus reducing the number of iterations needed to complete the coloring. Moreover, after 2 iterations, the random numbers of the nodes are recomputed. Figure 4c

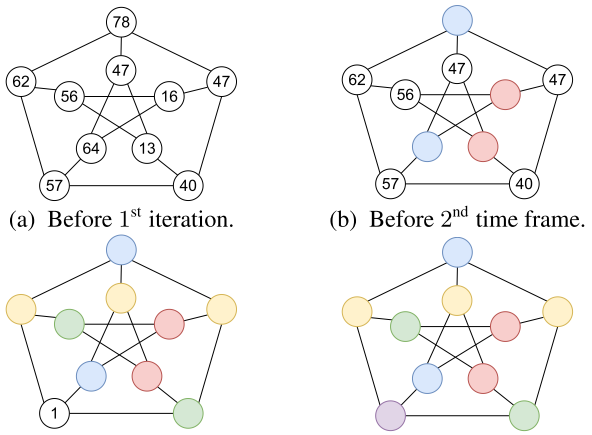


FIGURE 4. Application of the Gunrock implementation of the JPL algorithm on a small graph of 10 nodes.

shows the state of the graph before starting the 3rd iteration, after the recomputation is finished; however, only one node remains to be colored. The recomputation is inconsequential in this small example, but it helps in reducing the number of colors and iterations on larger graphs. The graph is colored using the same number of colors as in Figure 3, but using half the number of iterations rounded up.

IV. OUR COLORING PROCEDURE

Our implementation follows the Jones-Plassmann-Luby algorithm described in Section III-C. We designed our code to be compiled in two versions: One finding a single independent set for each iteration, and the other finding two independent sets. The two versions enable us to verify the speedup reported by Osama et al. [15], that can be achieved finding two independent sets per iteration. We will show in Section V that our speedup of the Min Max approach against the Max approach ranges between 1.5x and 2x on average. We declare the kernel function color_jpl, which takes as input the entire graph in CSR format, the pre-initialized array of random values *rand*, and the array where to store the colors. The total memory occupancy on the GPU is of $\big(4 \times (3n + m + 1)\big)$ bytes, assuming the architecture uses 32 bit integers. When working with graph instances that do not fit in the limited memory of the GPU, we partition the nodes of the graph in lexicographical order so that they fit in memory; thus, we perform multiple colorings to color each partition separately. The pseudo-code for the kernel function is shown in Algorithm 6.

To solve the problem of two neighboring nodes being assigned the same random value, as described in Section III-E, we proceed as follows. In the first phase, to obtain the array *rand* received as a parameter by the function color_jpl, we pre-generate a random permutation of a set of unique items such as V. We call this technique **value permutation**. Generating permutations is a simple

Algorithm 6 Our Implementation of the Jones-Plassmann-Luby Algorithm

color_jpl $(G = (V, E), rand, colors)$
1: $i \leftarrow 0$
2: $N \leftarrow V$
3: **while** $N \neq \emptyset$ **do**
4: $\quad c \leftarrow 2 * i + 1$
5: $\quad k \leftarrow 2 * i + 2$
6: $\quad$ **for** $v \in N$ in parallel on GPU **do**
7: $\quad\quad I^M \leftarrow I^M \cup \{v\}$
8: $\quad\quad I^m \leftarrow I^m \cup \{v\}$
9: $\quad\quad r_v \leftarrow rand(v + i \pmod n)$
10: $\quad\quad$ **for** $u \in (adj(v) \cap N)$ **do**
11: $\quad\quad\quad r_u \leftarrow rand(u + i \pmod n)$
12: $\quad\quad\quad$ **if** $r_v \leq r_u$ **then**
13: $\quad\quad\quad\quad I^M \leftarrow I^M \setminus \{v\}$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad\quad$ **if** $r_v \geq r_u$ **then**
16: $\quad\quad\quad\quad I^m \leftarrow I^m \setminus \{v\}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **end for**
20: $\quad$ **for** $v \in I^M$ in parallel on GPU **do**
21: $\quad\quad colors(v) \leftarrow c$
22: $\quad$ **end for**
23: $\quad$ **for** $v \in (I^m \setminus I^M)$ in parallel on GPU **do**
24: $\quad\quad colors(v) \leftarrow k$
25: $\quad$ **end for**
26: $\quad N \leftarrow N \setminus I^M$
27: $\quad N \leftarrow N \setminus I^m$
28: $\quad i \leftarrow i + 1$
29: **end while**

yet powerful way to solve the problem of two neighboring nodes being assigned the same random value. Thus, our *rand* array contains a permutation of V, unlike Gunrock's *rand* array, which includes random floating point numbers. To perform this task, we initially adopted the randomization function `std::shuffle`[2] available in the C++ standard library. Unfortunately, even though this function has linear complexity in the size of the array (i.e., the number of nodes in the graph), a close investigation of the entire execution time (not just the coloring phase), showed us that the computational overhead on large graphs could obfuscate the advantage of our approach. Although the array generation requires only the number of nodes of the graph to be performed, and its time could be entirely masked by other algorithmic phases performed in parallel (such as allocating, building, or loading the graph itself), we decided to investigate faster solutions. We reduced the generation time of the *rand* vector to a fraction of the original time by randomly generating the array using a custom `fast_rand`

[2]https://en.cppreference.com/w/cpp/algorithm/random_shuffle

function. The `fast_rand` function is a multiply-with-carry pseudo-random number generator, that allows to produce sequences of pseudo-random values with very long period, by using simple integer arithmetic logic. The function generates a weaker scattering of the generated values but this feature does not decrease the quality of the solution. This consideration may raise the question of how much quality of the randomness we can give away to speed up the vector generation process while not losing the quality and speed of the color computation. This area may be interesting for a future study on the subject. Moreover, inspired by our fix of the Gunrock implementation, we also change the random values assigned to each node every iteration. As in the Gunrock approach, regenerating the random values has a positive impact on the number of colors, since it helps split long chains of monotonic random values that would need many iterations to be colored. To avoid the overhead required by the generation of new arrays, we simulate a new permutation by accessing the same array circularly. In this way, the array still contains the same values, but each vertex v is assigned a new "random" value $rand_k(v)$ after k shifts, and the regeneration cost is meager. Shifting a n-element array in the device's global memory is an operation that requires synchronization between the threads of the grid, and programming a GPU to perform it requires some care. However, the random value of vertex v after k iterations, i.e., $rand_k(v)$, is the random value of vertex $(v + k \mod n)$ after 0 iterations, i.e., $rand_0(v + k \mod n)$. In other words, shifting an array by k positions to the left is equivalent to increment the index of the same amount k and performing the proper modulo operation to remain within the array's bound. We call this technique **index shift**, as it simulates an array circular shift by manipulating the index used to access the array. To simulate subsequent circular left shifts, one after each iteration, we decide to increase by one the indexes used by each thread cumulatively to access the *rand* array. Shifting the index does not add any significant overhead to the computation, as we do not write nor move data in memory. This technique is reported in Algorithm 6 on lines 9 and 11. Further experiments show that while the shift itself is always effective in reducing the number of colors; in some situations, this reduction can be further optimized by a full regeneration of the vector of random weights. By investigating the nature of these cases, we discovered that the probability that two adjacent nodes share an arc is higher than the probability that any pair of nodes share an arc. Although this proved to be valid only on some graphs, the non-correlation between nodes with neighboring indices could not be taken for granted. Given the nature of the problem, we increase the shift in the vector, such that the probability that the two nodes separated by a number of elements equal to the shift have a relationship is reduced considerably, Precisely as in the case of shifting a single element, this operation adds almost no computational cost, since it only changes the memory access address.

Section V shows that our approach reduces the number of iterations needed to compute the final coloring. Consequently, our code runs faster and solves all our benchmarks with fewer colors than all previous implementations. Moreover, our experiments unveil that shifts greater than four rarely give any benefit. Although this value has been evaluated experimentally, its meaning is the following. Shifting the vector of random numbers changes the relationships between the nodes possessing the various weights. An extremely simple analysis, which can be performed at almost no additional cost when reading the file, is calculating the highest number of consecutive nodes that form adjacency chains. A shift of a length greater than this chain would allow us to avoid having the same random number assigned to another element of the same chain. However, this would still be a superficial analysis, since the values don't need to leave the chain in which they are located, as it is sufficient to vary the configuration of the adjacent nodes. Consequently, the size of the shift can be expressed as a function of the size of the graph and how dense or sparse it is.

Figure 5 shows how our implementation of the JPL algorithm colors the same small graph used in all other examples. The numbers we permute after each iteration, the ones stored in the array *rand*, are displayed inside their corresponding node v. The numbers are unique and included in the range of integers between 0 and 9. Figures 5b, 5c, and 5d show the state of the graph after each iteration of the algorithm. The unique numbers move according to the technique of index shifting. In the picture, we assume the nodes are ordered clockwise, outer to inner, starting from the top-most node. Each permutation moves those numbers corresponding to the circular left shift of the array. The final solution of Figure 5d is congruent with the one obtained by the original Gunrock implementation (and represented in Figure 4), but the algorithm runs faster and uses fewer colors, as we illustrate in the next section.

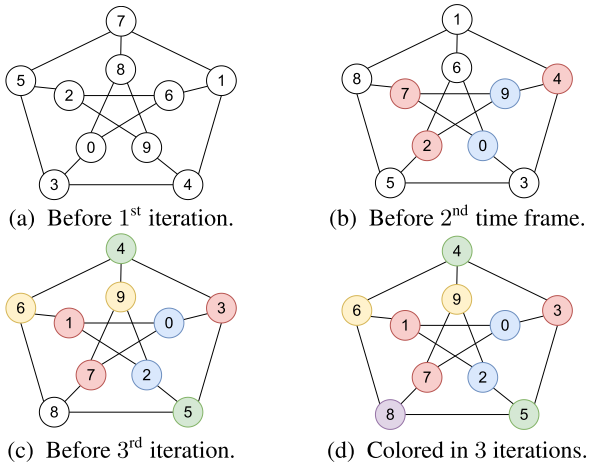


FIGURE 5. Application of our implementation of the JPL algorithm on a small graph of 10 nodes.

TABLE 1. The main characteristics of the benchmark graphs used during our experimental analysis. The graphs are numbered from 1 to 28 to find an easy correspondence in the following plots. Column *Type* indicates the main characteristics of each graph: Real (r) or generated (g), undirected (u) or directed (d), following a power-law degree distribution (p) or not (-). Notice that power-law graphs, which have distinct characteristics and on which the different approaches behave differently, have been inserted in the second part of the table.

#	Graph	Nodes	Edges	Avg degree	Type
1	af_shell3	504,855	17,588,875	34.8	ru-
2	apache2	715,176	4,817,870	6.7	ru-
3	ecology2	999,999	4,995,991	5.0	ru-
4	G3_circuit	1,585,478	7,660,826	4.8	ru-
5	offshore	259,789	4,242,673	16.3	ru-
6	parabolic_fem	525,825	3,148,801	6.0	ru-
7	thermal2	1,228,045	8,580,313	7.0	ru-
8	ASIC_320ks	321,671	1,827,807	5.7	rd-
9	atmosmodd	1,270,432	8,814,880	6.9	rd-
10	cage13	445,315	7,479,343	16.8	rd-
11	FEM_3D_thermal2	147,900	3,489,300	23.6	rd-
12	thermomech_dK	204,316	2,846,228	13.9	rd-
13	rgg_n_2_15_s0	32,768	320,480	9.8	gu-
14	rgg_n_2_16_s0	65,536	684,254	10.4	gu-
15	rgg_n_2_17_s0	131,072	1,457,506	11.1	gu-
16	rgg_n_2_18_s0	262,144	3,094,566	11.8	gu-
17	rgg_n_2_19_s0	524,288	6,539,532	12.5	gu-
18	rgg_n_2_20_s0	1,048,576	13,783,240	13.1	gu-
19	rgg_n_2_21_s0	2,097,152	28,975,990	13.8	gu-
20	rgg_n_2_22_s0	4,194,301	60,718,396	14.5	gu-
21	rgg_n_2_23_s0	8,388,608	127,002,786	15.1	gu-
22	rgg_n_2_24_s0	16,777,216	265,114,400	15.8	gu-
23	qg.order100 [21]	10,000	1,980,000	198.0	gd-
24	twitch_gamers [22]	168,114	13,595,114	80.9	rup
25	email_Enron	36,692	367,662	10.0	rdp
26	hollywood-2009	1,139,905	112,751,422	98.9	rup
27	indichina-2004	7,414,866	301,969,638	40.7	rdp
28	soc-LiveJournal1	4,847,571	85,702,474	17.7	rdp

V. EXPERIMENTAL RESULTS

We perform our experiments on an i9 10900KF CPU running at 3.7 GHz, with 10 cores, 20 threads, and 64 GB of RAM, coupled with a GPU NVIDIA RTX 3070 with 5888 CUDA cores and 8 GB of dedicated memory. The operating system is Linux Ubuntu 22.04.1 LTS. The code was compiled using Clang version 16.0.4 for the CPU implementations, and NVIDIA's CUDA Compiler (NVCC) version 12.0 for the GPU programs.

We run the coloring implementations described in Sections III and IV on the set of graphs reported in Table 1. The set contains the same graphs used by Osama et al. [15], plus some extra graphs, namely, email_Enron, twitch_gamers, qg.order100, hollywood-2009, indochina-2004, and soc-LiveJournal1 For each graph, the table reports the number of vertices and edges. Column *Type* indicates whether the graphs are real (**r**) or generated (**g**), and undirected (**u**) or directed (**d**). Moreover, the last part of the table includes graphs showing a power-law degree distribution (**p**). These five graphs marked with letter (**p**) are kept separated because the coloring algorithms have different behavior (and performances) on these instances. We gathered the graphs through the Sparse Matrix Collection website [20], if not otherwise specified.

If we consider the larger test graph, i.e., rgg_n_2_24_s0, with more than 16 million nodes, our implementation requires a GPU memory of

$$\begin{aligned} &4 \times (3n + m + 1) \\ &= 4 \times (3 \times 16,777,216 + 265,114,400 + 1) \\ &\approx 1.18 \text{ GB} \end{aligned}$$

As a consequence, memory is not an issue, as the most extensive graph occupies less than 15% of the total memory available in our GPU. To take into consideration runtime fluctuations and provide a better estimate of the number of colors used by all non-deterministic algorithms, we run each implementation 20 times on each graph. Thus, our tables report the average time spent coloring the graph and the average number of colors used, rounded to the nearest integer.

Table 2 reports the coloring times (in milliseconds) for each graph, and the following algorithms:

- $\text{GM}_{\text{s-imp}}$, our improved synchronous version of Gebremedhin-Manne (Algorithm 3), running on the CPU.
- $\text{GM}_{\text{a-std}}$, our standard asynchronous version of Gebremedhin-Manne (Algorithm 2), running on the CPU.
- $\text{JPL}_{\text{min-max}}$, our implementation of the JPL min-max procedure (Algorithm 4) executing on CPU.
- cuSparse [10], i.e, the csrColor Cohen and Castonguay procedure (Section III-D), running on the GPU.
- Gunrock [15], Gunrock's algorithm (Algorithm 5), running on the GPU,
- Atos [16], a custom implementation of GM on GPU originally running on Volta architectures.
- JPL_{max} and $\text{JPL}_{\text{min-max}}$, our index-shift implementations (Algorithm 6) running on the GPU.

Notice that Table 2 considers only the coloring times and ignores all pre-processing and post-processing overheads, as done by the authors of all other approaches. We present results including pre- and post-processing times (comprising transfer times) in Table 3.

For the majority of the graphs, our two implementations of the Gebremedhin-Manne algorithm, running on parallel CPU, are slower compared to the implementations (both our own and state-of-the-art) executing on the GPU. The synchronous implementation is between 1 and 4 orders of magnitude slower than the best result we obtain on the GPU, while the asynchronous implementation is between 0 and 2 orders of magnitude slower. However, it is interesting how these slower implementations, especially the asynchronous one, perform comparably or even better on specific graphs, such as, twitch_gamers, email_Enron and qg.order100, hollywood-2009, indochina-2004, and soc-LiveJournal1, which for the most part are the ones to show a power-law degree distribution. The Cohen-Castonguay implementation, which on the other graphs shows the worst

TABLE 2. Average coloring time for each one of our implementations. Columns' headers have the meaning described in the itemization included in the main text. On the CPU-side, we indicate with $GM_{s\text{-}imp}$ and $GM_{a\text{-}std}$ our implementations of the GM synchronous and asynchronous algorithm, and with $JPL_{min\text{-}max}$ the JPL procedure. On the GPU side, we report the results of cuSparse, Gunrock, and Atos. The last two columns include our implementations on GPU. Once more, the graphs after the horizontal line (used as a separator) follow a power-law degree distribution.

#	Graph	CPU-based			GPU-based				
					State-of-the-art			Our Methods	
		$GM_{s\text{-}imp}$	$GM_{a\text{-}std}$	$JPL_{min\text{-}max}$	cuSparse	Gunrock	Atos	JPL_{max}	$JPL_{min\text{-}max}$
1	af_shell3	1387.70	63.39	29.31	5.97	28.09	12.15	10.14	5.63
2	apache2	1888.60	30.10	7.67	3.89	1.19	12.30	1.67	0.32
3	ecology2	2679.50	36.73	6.99	3.22	0.87	16.56	1.54	0.32
4	G3_circuit	4267.00	58.88	9.06	4.16	1.18	52.09	1.97	0.49
5	offshore	695.62	18.70	9.36	3.48	3.29	3.93	1.68	0.98
6	parabolic_fem	1371.00	22.76	5.94	2.79	1.06	0.84	1.22	0.43
7	thermal2	3285.30	59.97	9.37	4.52	2.03	3.83	2.00	0.88
8	ASIC_320ks	833.46	13.29	5.17	3.75	3.42	0.93	3.23	1.84
9	atmosmodd	3434.60	58.63	10.76	4.57	1.72	20.48	2.32	0.56
10	cage13	1182.90	32.68	15.14	4.70	5.72	2.47	2.78	1.76
11	FEM_3D_thermal2	396.37	13.58	11.85	4.29	3.48	6.03	1.87	0.84
12	thermomech_dK	540.76	13.36	6.56	3.44	2.04	1.42	1.09	0.73
13	rgg_n_2_15_s0	87.68	2.14	4.06	3.11	0.60	0.89	0.37	0.30
14	rgg_n_2_16_s0	174.27	3.90	4.71	3.20	0.76	1.02	0.47	0.31
15	rgg_n_2_17_s0	342.79	7.93	5.83	3.25	1.16	1.50	0.73	0.37
16	rgg_n_2_18_s0	684.74	15.60	8.24	4.45	2.10	2.39	1.18	0.72
17	rgg_n_2_19_s0	1397.50	31.94	11.53	3.91	4.43	4.47	3.03	0.78
18	rgg_n_2_20_s0	2806.80	66.06	19.35	5.71	8.38	7.95	5.02	2.10
19	rgg_n_2_21_s0	5814.40	139.64	32.26	8.68	16.90	16.70	9.83	4.23
20	rgg_n_2_22_s0	11988.00	285.03	56.44	15.29	35.85	35.21	18.95	12.14
21	rgg_n_2_23_s0	24789.00	592.69	113.59	38.11	77.22	70.82	45.29	19.64
22	rgg_n_2_24_s0	52785.00	1221.10	230.46	75.73	167.91	140.41	103.40	45.07
23	qg.order100	51.98	15.65	98.24	12.99	35.51	3.94	21.61	10.66
24	twitch_gamers	525.55	51.39	143.25	75.60	2774.60	2.54	2112.30	1012.20
25	email_Enron	99.69	3.16	19.98	9.50	36.74	1.48	23.30	12.79
26	hollywood-2009	80926.00	579.20	1250.26	268.14	8526.83	69.42	7215.19	3263.75
27	indochina-2004	1266230.00	4105.95	2382.04	1450.90	55313.78	642.28	32072.10	14824.90
28	soc-LiveJournal1	7954.65	593.27	648.35	98.61	2083.13	119.15	1910.77	1003.30

performance on the GPU, performs the best on these six graphs out of the GPU algorithms. To better understand this peculiar behavior, we analyze the topology of these graphs. All graphs share a large maximum degree: twitch_gamers has a maximum degree of 35279, email_Enron of 1383, hollywood-2009 of 11467, indochina-2004 of 256425, soc-LiveJournal1 of 20333, and all nodes of qg.order100 have a degree of 198. Furthermore, all graphs, except for qg.order100, have a power-law degree distribution, which implies the presence of a minimal number of nodes with an enormous number of edges. These degrees are very high when compared to the other benchmark graphs; among the others, the highest degree is presented by ASIC_320ks, which has a maximum degree of 412, but an average degree of 5.7, meaning that the majority of its nodes have a much lower degree. On the implementations of the JPL algorithm for GPU, including the Gunrock implementation, nodes with these large degrees cause many issues of memory read instructions inside the loops on line 10 of Algorithm 6 and on line 14 of Algorithm 5, to fetch the random number associated with each neighbor. On graphs twitch_gamers and email_Enron, where node degrees vary, this also causes load imbalance, as threads assigned to color small-degree nodes are idle, while threads coloring large-degree nodes take a longer time to check all the neighbors. As the process of checking all neighbors is repeated at each iteration until the node with the most significant degree is colored, the total runtime of the algorithm becomes longer.

On the other hand, the Cohen-Castonguay algorithm does not suffer as much when running on these graphs because it computes the random values at runtime using hash functions, and the implementation assigns a larger number of colors for each iteration, ultimately completing coloring in fewer iterations. Similarly to the Gebremedhin-Manne CPU implementation, Atos shows excellent performance on these graphs. As Atos can assign a color reading the list of neighbors only once for each node, it potentially creates many conflicts; however, it also severely improves the overall performance when it converges faster. We can see this behavior both on the GPU and the CPU, except for the graph indochina-2004 in which the JPL CPU implementation outperforms the GM program. However, this is due to the poor thread scheduling that causes a huge oscillation in execution time over multiple runs, negatively impacting the average performances. GM-based methods still outperform JPL-based ones on this graph in the best case.

Table 3 compares wall-clock times considering the entire process, comprised of the pre-processing, coloring, and post-processing phases. The table illustrates the impact of our array generation process and the transfer time (to and from

TABLE 3. Detailed comparison of our JPL min-max approach against Gunrock. Notice that Gunrock pre-processing times are roughly equivalent to the sum of our times, including the vector randomization, the GPU allocation, and the CPU-to-GPU transfer time. The post-processing phase includes the GPU-to-CPU transfer time.

#	Graph	Gunrock				$JPL_{min\text{-}max}$						Speedup
		Preprocess	Process	Postprocess	Total	Preprocess			Process	Postprocess	Total	
						Randomization	Allocation	Transfer				
1	af_shell3	74.74	28.09	6.04	108.87	2.91	0.34	6.44	5.50	0.24	15.43	7.06
2	apache2	57.60	1.19	6.54	65.33	4.10	0.26	2.23	0.39	0.32	7.30	8.95
3	ecology2	51.02	0.87	8.95	60.84	5.75	0.26	2.51	0.36	0.41	9.29	6.55
4	G3_circuit	58.78	1.18	13.93	73.89	9.01	0.31	3.95	0.57	0.61	14.45	5.11
5	offshore	53.98	3.29	2.95	60.22	1.50	0.28	1.76	0.82	0.17	4.53	13.29
6	parabolic_fem	50.16	1.06	4.99	56.20	2.96	0.27	1.66	0.38	0.26	5.53	10.16
7	thermal2	52.59	2.03	10.77	65.39	7.02	0.32	4.78	0.74	0.59	13.45	4.86
8	ASIC_320ks	58.98	3.42	3.11	65.50	1.81	0.27	0.94	1.29	0.20	4.51	14.53
9	atmosmodd	53.23	1.71	11.76	66.71	7.31	0.31	4.00	0.64	0.51	12.77	5.22
10	cage13	53.14	5.72	5.20	64.07	2.48	0.25	2.94	1.60	0.24	7.51	8.53
11	FEM_3D_thermal2	49.73	3.48	2.02	55.23	0.86	0.27	1.32	0.95	0.10	3.50	15.78
12	thermomech_dK	54.49	2.04	2.32	58.85	1.17	0.27	1.19	0.51	0.14	3.28	17.94
13	rgg_n_2_15_s0	53.26	0.60	0.49	54.35	0.20	0.27	0.20	0.19	0.03	0.89	61.07
14	rgg_n_2_16_s0	61.82	0.76	0.76	63.34	0.39	0.27	0.35	0.24	0.05	1.30	48.72
15	rgg_n_2_17_s0	48.31	1.16	1.27	50.74	0.82	0.28	0.66	0.37	0.09	2.22	22.86
16	rgg_n_2_18_s0	61.40	2.10	2.71	66.21	1.59	0.28	1.39	0.63	0.17	4.06	16.31
17	rgg_n_2_19_s0	51.60	4.43	5.04	61.08	3.18	0.28	2.88	1.11	0.26	7.71	7.92
18	rgg_n_2_20_s0	69.09	8.38	10.06	87.53	6.38	0.49	6.25	2.24	0.45	15.81	5.54
19	rgg_n_2_21_s0	71.55	16.90	18.47	106.92	12.70	0.40	11.64	4.69	0.79	30.22	3.54
20	rgg_n_2_22_s0	105.89	35.85	32.93	174.66	23.34	0.61	24.03	9.79	1.52	59.29	2.95
21	rgg_n_2_23_s0	149.53	77.22	62.62	289.37	38.21	0.94	49.69	22.38	2.88	114.10	2.53
22	rgg_n_2_24_s0	246.17	167.91	122.67	536.76	60.34	1.59	81.43	52.24	6.38	201.98	2.65
23	qg.order100	49.59	35.51	0.64	85.75	0.06	0.27	0.74	9.01	0.01	10.09	8.50
24	twitch_gamers	66.67	2774.63	3.77	2845.08	1.00	0.32	4.97	1012.45	0.11	1018.85	2.79
25	email-Enron	62.16	36.74	0.49	99.39	0.25	0.26	0.18	14.56	0.03	15.28	6.50
26	hollywood-2009	141.02	8086.35	12.41	8239.78	0.26	0.75	38.61	3510.13	0.47	3550.21	2.32
27	indochina-2004	273.73	58306.92	57.27	58637.92	39.28	1.70	104.76	15933.30	2.52	16081.56	3.65
28	soc-LiveJournal1	119.12	2230.96	39.58	2389.66	1.08	0.68	30.56	1066.01	1.67	1100.00	2.17

the GPU) on the overall execution time. The time required by our algorithm is divided into five steps such that the only time ignored are the ones required to read and write the graph to disk. The first three execution steps can be merged in Gunrock's pre-process time. Even though the transfer time is usually more significant than the execution time, we can still show that our implementation is able to solve all of our instances in less time than Gunrock. Speedups vary from 2.17x to 61.07x with a geomean speedup of 7.43x. Moreover, please notice that, to the best of our knowledge, the transfer time is rarely considered in literature when comparing CPU and GPU performances. After all, it is more an architectural issue than an algorithmic one, and it can currently be reduced by compression and decompression strategies, the Zero-Copy memory approach, or by reading the data directly from disk, which are methodologies available on the newest NVIDIA cards. Furthermore, in an environment in which CPUs and GPUs collaborate to solve a set of problems, it is unclear whether the memory transfer time has to be "added" to the GPU and not to the CPU performances.

Notice that we show the transfer times for our $JPL_{min\text{-}max}$ approach, but at the same time, our measurements are also valid for JPL_{max}. The only difference between the two is the coloring time. Moreover, we put effort into optimizing the transfer times using CUDA Streams. This optimization allowed us to reduce the memory transfer costs and avoid useless synchronizations that proved to be small bottlenecks. However, as we used CUDA streams to gather precise timings, we performed GPU-CPU synchronizations after each phase. Furthermore, for this comparison, we add all of the times together to compute the speedup; operations like the randomization of the weights vector that is performed on the GPU using the *fast_rand* function detailed before can be performed while reading the graph itself from the disk since the number of nodes is known from the beginning of the process.

Figure 6 plots the speedups of all GPU implementations over the Gunrock implementation. We evaluate the ratio between the computation time of the Gunrock strategy and all other methods X, i.e., $t(Gunrock)/t(X)$, and displayed these values on a logarithmic scale on the y-axis.

Obtaining the minimum and the maximum speedups with our JPL implementations on the same graphs is not coincidental. The two implementations are coded such that $JPL_{min\text{-}max}$ should color twice as many nodes as JPL_{max}; thus, it is not surprising that the processing stage is twice as fast on the same graph structure. In Figure 7, we display the speedup obtained by coloring two independent sets per iteration ($JPL_{min\text{-}max}$) over the standard approach of coloring a single one (JPL_{max}).

To better understand the differences between our $JPL_{min\text{-}max}$ implementation and the state-of-the-art implementation from the Gunrock library, we use the Nsight Compute profiler to collect information on their runtimes.

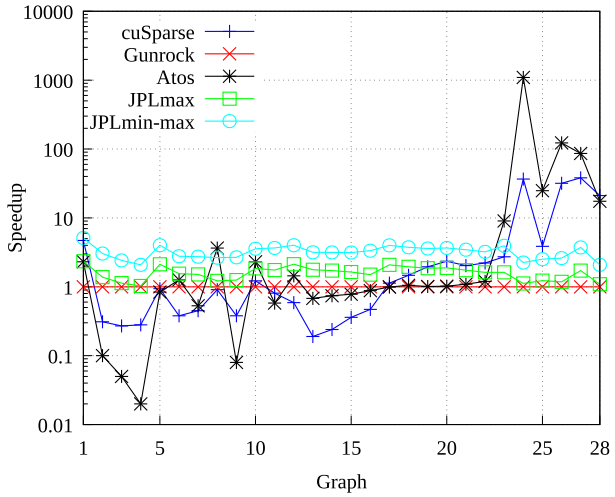


FIGURE 6. Speedups of our implementations JPL_{max} and $JPL_{min\text{-}max}$ against CuSparse, Gunrock, and Atos. The Gunrock procedure (in red color) is used as a reference and normalized to one.

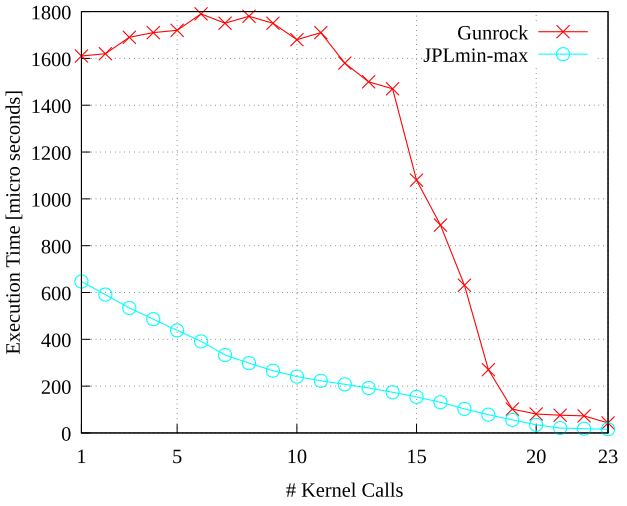


FIGURE 8. Elapsed time to complete each kernel launch within our $JPL_{min\text{-}max}$ strategy and the one delivered by Gunrock.

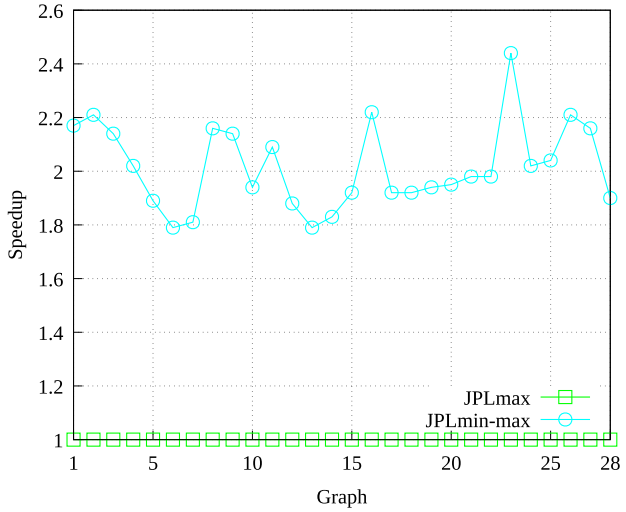


FIGURE 7. Speedup of our $JPL_{min\text{-}max}$ approach (dealing with two independent sets for each iteration) over our JPL_{max} methodology (dealing with a single independent set). The expected 2x factor is reached on average as overheads are negligible.

Nsight Compute collects data on every kernel launched during the execution. Some of the metrics collected include grid dimensions, execution time, the average number of threads active per warp (to estimate divergence), cache hit rate, and many more. From the profiles, we know that both implementations rely on more than one kernel to perform the coloring. For $JPL_{min\text{-}max}$, the main kernel that actively performs the coloring (*color_jpl_kernel*) is followed by two auxiliary kernel calls (*DeviceReduceKernel* and *DeviceReduceSingleTileKernel*) used to compute the number of uncolored nodes after every iteration. For the Gunrock implementation, the coloring is performed by the kernel named *Kernel*, and auxiliary tasks are performed by the kernels named *GetEdgeCounts*, *launch_box_cta_k*, and *gen_sequenced*. Among these kernels, *gen_sequenced* is called once every two calls of *Kernel*, and it is the kernel that regenerates the array of random numbers. Since the execution time of the auxiliary kernels in both algorithms is negligible compared to the execution time of the main kernels, we do not consider them, as they have a limited impact on the overall execution time. In Figure 8, we represent the execution times of the main kernels (*color_jpl_kernel* for $JPL_{min\text{-}max}$ and *Kernel* for Gunrock/color) during the coloring of the graph *af_shell3*.

Figure 8 shows that each new kernel launched by the $JPL_{min\text{-}max}$ implementation terminates its execution slightly faster than the previous kernel. The first kernel achieves the maximum execution time, taking 647 μs, while the faster kernel is the last one, terminating its execution in 16 μs. The execution times of the kernels run by the Gunrock/color implementation can be divided into two phases. In the first phase, encompassing the first 14 kernel launches, the execution times oscillate around the value of 1669 μs, with a maximum of 1790 μs for the 6^{th} kernel, and a minimum of 1470 μs for the 14^{th} kernel. The second phase, spanning from the 15^{th} kernel launch up to the last one, approximately follows a negative exponential trend, going from a maximum of 1080 μs for the 15^{th} kernel to a minimum of 43 μs for the 23^{rd} and last kernel launch. Some caching issue likely causes the discrepancy in the two phases of the Gunrock implementation. Indeed, the kernel is written such that the random value associated with the current node is not cached, and needs to be read multiple times inside the loop on lines 15 and 18 of Algorithm 5. Multiple reading operations cause extremely high execution times for the first iterations, which rapidly drop in the second phase, after the majority of the nodes have been colored. On the other hand, our $JPL_{min\text{-}max}$ implementation does not suffer from this problem, as the random values are cached in registers on lines 9 and 11 of Algorithm 6. Our analysis of kernel running times also includes the version of our $JPL_{min\text{-}max}$ that does

not implement index shifting. We do not report those runtimes in Figure 8 as they are very similar and follow the same trend as the ones reported for strategy $\text{JPL}_{\text{min-max}}$. The version without index shifting is on average 13% faster on the first 16 kernels, and around 175% slower on the remaining 7 kernel calls. However, the version without index shifting terminates only after 32 kernel runs, meaning that more colors are used in the solution.

Table 4 reports the average number of colors (over 20 runs and rounded to the nearest integer) used by our implementations over all our test graphs. The data shows how the two CPU implementations of the Gebremedhin-Manne algorithm (namely, $\text{GM}_{\text{s-imp}}$ and $\text{GM}_{\text{a-std}}$) consistently generate solutions using fewer colors than the GPU implementations. As the Atos approach is based on GM, this is also true for Atos. Between the two GM implementations, the improved version uses fewer colors in all graphs other than apache2 and ecology2. This behavior is expected as the improved algorithm is formulated to reduce the number of colors generated during the first coloring step of the algorithm. However, since the coloring is performed non-deterministically, the final improved solution is not guaranteed to use fewer colors than the standard solution. Figure 9 uses the synchronous improved GM implementation as a baseline to compare the number of colors of all other methods. The figure reports the data of Table 4 as a percentage increase, computed as $\left(\left(c(X) - c(GM_{s-imp}\right)/c(X)\right)$.

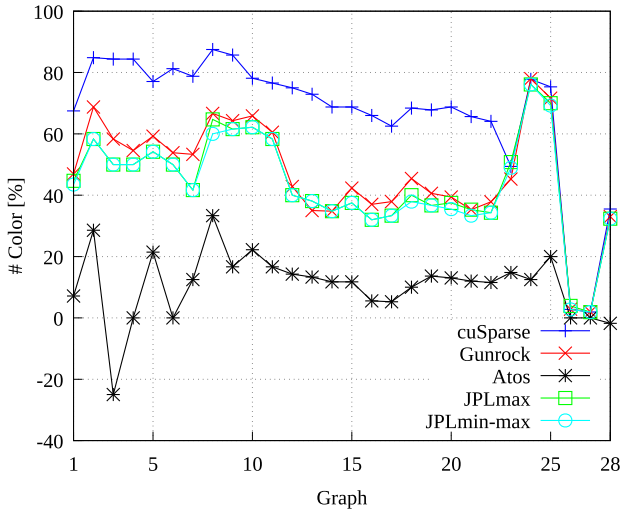
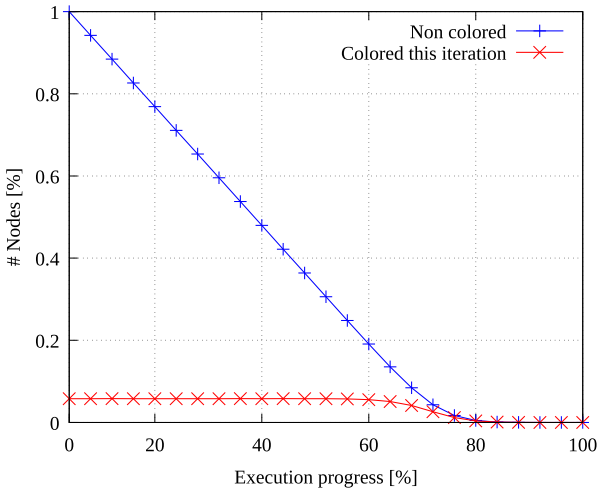
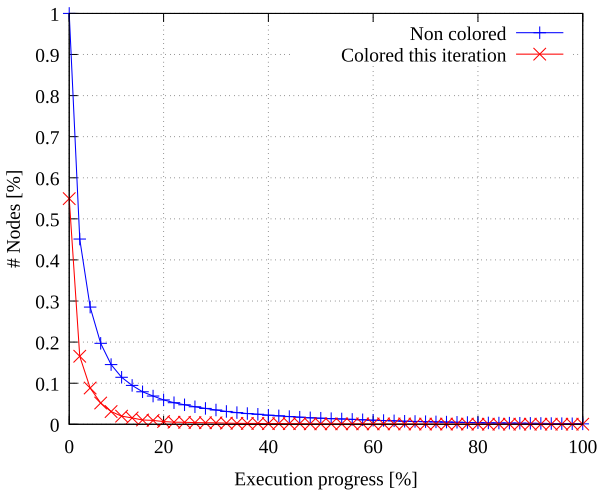


FIGURE 9. Percentage variations in the number of colors used by the different GPU-based methods with respect to $\text{GM}_{\text{s-imp}}$ used as a reference and CPU-based.

On all graphs other than qg.order100, the state-of-the-art implementation of the Cohen-Castonguay algorithm is the one to generate the most colors, with an average percentage over $\text{GM}_{\text{s-imp}}$ of 55.5%. The three other implementations manage to use fewer colors. Our $\text{JPL}_{\text{min-max}}$ implementation uses 7% fewer colors on average than the state-of-the-art implementation of the same algorithm on Gunrock.



(a) A balanced behavior on a graph including vertices with low or average degree.



(b) An unbalanced behavior on a graph including a few vertices with a very high degree.

FIGURE 10. The y-axis represents a percentage of the nodes, whereas the x-axis represents the execution progress.

Our JPL_{max} implementation shows mixed results, using more colors than the Gunrock implementation in some graphs and matching the colors of $\text{JPL}_{\text{min-max}}$ on other graphs, including the rgg graph family. JPL_{max} reports an average percentage over $\text{GM}_{\text{s-imp}}$ of 46.17%, Gunrock of 39.24%, and 36.39% for $\text{JPL}_{\text{min-max}}$.

A. PERFORMANCE ANALYSIS

To study how our algorithms face load imbalance and variable-size frontier sets, we present the following analysis.

Figure 10a represents the percentage of the nodes colored at each iteration, and the ones which remain uncolored, as a function of the number of main coloring iterations. We report these values for the graph *rgg_n_2_24_s0*, but this behavior is typical to all graphs on which our algorithm performs

TABLE 4. The average number of colors for each one of our implementations. On the CPU side, we present our implementations of the GM synchronous and asynchronous algorithm ($GM_{s\text{-}imp}$ and $GM_{a\text{-}std}$, respectively), and the JPL procedure ($JPL_{min\text{-}max}$). On the GPU side, we report the results of cuSparse, i.e., the csrColor Cohen and Castonguay implementation, Gunrock, i.e., the Gunrock's algorithm, and Atos from Chen et at. The last two columns include our implementations on GPU.

#	Graph	CPU-based			GPU-based				
					State-of-the-art			Our Methods	
		$GM_{s\text{-}imp}$	$GM_{a\text{-}std}$	$JPL_{min\text{-}max}$	cuSparse	Gunrock	Atos	JPL_{max}	$JPL_{min\text{-}max}$
1	af_shell3	26	27	46	80	49	28	47	46
2	apache2	5	4	13	33	16	7	12	12
3	ecology2	5	4	10	32	12	4	10	10
4	G3_circuit	5	5	10	32	11	5	10	10
5	offshore	11	13	23	48	27	14	24	24
6	parabolic_fem	6	6	12	32	13	6	12	12
7	thermal2	7	7	12	33	15	8	12	12
8	ASIC_320ks	6	8	15	48	18	9	17	15
9	atmosmodd	5	6	13	35	14	6	13	13
10	cage13	14	16	37	64	41	18	37	37
11	FEM_3D_thermal2	15	18	36	64	38	18	36	36
12	thermomech_dK	12	13	20	48	21	14	20	20
13	rgg_n_2_15_s0	13	14	21	48	20	15	21	21
14	rgg_n_2_16_s0	15	17	23	48	23	17	23	23
15	rgg_n_2_17_s0	15	16	24	48	26	17	24	24
16	rgg_n_2_18_s0	17	19	25	50	27	18	25	25
17	rgg_n_2_19_s0	18	19	27	48	29	19	27	27
18	rgg_n_2_20_s0	18	19	29	57	33	20	30	29
19	rgg_n_2_21_s0	19	20	30	59	32	22	30	30
20	rgg_n_2_22_s0	20	22	31	64	33	23	32	31
21	rgg_n_2_23_s0	22	24	33	64	34	25	34	33
22	rgg_n_2_24_s0	23	23	35	64	37	26	35	35
23	qg.order100	121	147	238	239	221	142	246	237
24	twitch_gamers	112	118	468	504	509	132	469	469
25	email_Enron	36	42	118	146	127	45	120	117
26	hollywood-2009	2209	2209	2287	2272	2274	2209	2299	2272
27	indochina-2004	6849	6849	6985	6983	6986	6849	6983	6982
28	soc-LiveJournal1	347	340	513	538	518	341	513	513

optimally. The number of nodes colored during each step remains practically constant, and it drops only when the execution is ending, as the remaining uncolored nodes are only a tiny fraction of the originals but they still require multiple passes to be colored. This behavior is characteristic of an efficient solution and indicates that our algorithm works at its best and is usually much faster than all competitors.

On the contrary, Figure 10b represents a graph with power-law degree distribution, more specifically, indochina-2004. However, as for the previous analysis, this behavior is ubiquitous for all graphs of this type. This graph has a conformation for which most nodes can be colored in very few passes, as they have very few neighbors and are located toward the graph's edges. The remaining nodes are those in the most populated areas and require numerous iterations to be successfully colored. Although it is possible to imagine a parallelism between the number of uncolored nodes and the size of the search frontier, this is not appropriate for the JPL approach, which, at each iteration, checks all nodes by skipping those that have already been colored. This factor implies that the size of the frontier is constant throughout the execution. As a consequence, the number of threads that need to operate after a node is received decreases as the number of iterations advances. This consideration, in turn, increases the load imbalance and the divergence. In particular, this is true for graphs that follow the power law. The NVIDIA Nsight Compute shows 20 active threads per warp on average for rgg_n_2_24_s0 but only 12 for indochina-2004 (and with a smaller number of instructions issued per cycle). Even the warp occupancy shows an imbalanced workload, as it is equal to 90% in the first case and 24% in the second one. These values show that the higher the number of steps required by the JPL procedure, the more expensive the operation becomes. On the other hand, Atos is more efficient for this graph structure, as it has over a 72% warp occupancy on average on indochina-2004, showing better use of the threads on the GPU.

VI. CONCLUSION

This paper describes, studies, and implements the most efficient state-of-the-art graph coloring algorithms running either on multi-core CPUs or many-core GPUs. We put particular attention to the algorithm by Luby, Jones, and Plassmann, which improves the algorithm efficiency by coloring independent sets of vertices.

We present two GPU implementations of this algorithm, which differ in the number of independent sets colored at each iteration. We enhanced these implementations with ''value permutation'', a method to generate a random permutation of a set of unique items, and ''index shifting'', a technique to simulate a circular array shift with a meager cost compared to the original strategies. These techniques improve the runtime of the algorithms, and they also reduce the number of colors used for coloring a graph.

We compared our implementations with three state-of-the-art implementations of graph coloring, namely, NVIDIA's

cuSparse, Gunrock, and Atos. As far as the pure coloring procedure is concerned (without pre- and post-processing), we show that our fastest implementation presents geomean (harmean) speedups of 3.16x (3.05x) against Gunrock, 4.09x (3.06x) against cuSparse, and 4.45x (2.21x) against Atos on mesh-like graphs. When we concentrate on the entire process (pre-processing, processing, and post-processing phases, including transfer times) our implementation has geomean (peak) speedups of 7.43x (61.07x) against Gunrock (the fastest of the competitors). At the contrary, the algorithm performs significantly worse when applied to scale-free graphs, where it is competitive only against Gunrock, the other implementation of the JPL algorithm. It shows a geometric mean (harmonic mean) of 2.76x (2.71x) against Gunrock, 0.13x (0.11x) against cuSparse, and 0.03x (0.01x) against Atos. At the same time, our approach can generate solutions using less colors than the other JPL-based procedures. With graphs that contain vertices with a huge number of arcs, our procedure (as all other JP-based algorithms) is slower than GM-based procedures and Atos. Since computing the characteristics of a given graph is a task that can be performed while reading or storing it, it is consequently possible to use a multi-engine approach and select the best algorithm to solve each instance as quickly as possible.

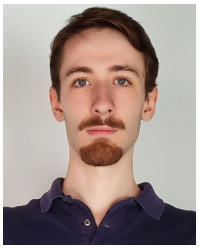
Further research is needed to study how our implementation can be further improved. Indeed, it is interesting to notice how our index shift technique stemmed from our initial decision to use the value permutation strategy to obtain unique numbers. Gathering random numbers from a uniform distribution would ultimately incur in a too high overhead to be recomputed. For this reason, we encourage further research to change the variables at play in an algorithm. Moreover, it would also be beneficial to implement other algorithms on many-core GPU architectures, as the speedup provided by those devices is substantial but graph algorithms rely on a lot on memory operations and researchers have been unable to exploit their power completely with graphs.

Our work shares many of the limitations common to other GPU algorithms. In particular, some time is required to load all the data needed for the program execution on the GPU itself. Moreover, tiny frontiers and unbalanced load problems can still be found. Similarly to Gunrock, we must synchronize all threads on the device after each coloring phase, introducing significant delays as the number of colors increases. Moreover, we show that our implementation of the GPU JPL algorithm shares the same weaknesses with the CPU algorithm over the coloring quality. Performance-wise, the duration of the computation strongly depends on thread divergence, which tends to increase with the number of coloring iterations.

REFERENCES

[1] F. T. Leighton, "A graph coloring algorithm for large scheduling problems," *J. Res. Nat. Bur. Standards*, vol. 84, no. 6, p. 489, Nov. 1979.
[2] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *Comput. J.*, vol. 10, no. 1, pp. 85–86, Jan. 1967.
[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Lang.*, vol. 6, no. 1, pp. 47–57, Jan. 1981.
[4] F. Akman, "Partial chromatic polynomials and diagonally distinct Sudoku squares," 2008, *arXiv:0804.0284*.
[5] K. Giaro, M. Kubale, and P. Obszarski, "A graph coloring approach to scheduling of multiprocessor tasks on dedicated machines with availability constraints," *Discrete Appl. Math.*, vol. 157, no. 17, pp. 3625–3630, Oct. 2009.
[6] M. Garey and D. Johnson, *uters and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.
[7] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency, Pract. Exp.*, vol. 12, no. 12, pp. 1131–1146, 2000.
[8] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, vol. 15, no. 4, pp. 1036–1053, Nov. 1986.
[9] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 654–669, May 1993, doi: 10.1137/0914041.
[10] J. Cohen and P. Castonguay, "Efficient graph matching and coloring on the GPU," in *Proc. GPU Technol. Conf.*, 2012, pp. 1–10.
[11] A. Garbo and S. Quer, "A fast MPEG's CDVS implementation for GPU featured in mobile devices," *IEEE Access*, vol. 6, pp. 52027–52046, 2018.
[12] G. Cabodi, P. Camurati, A. Garbo, M. Giorelli, S. Quer, and F. Savarese, "A smart many-core implementation of a motion planning framework along a reference path for autonomous cars," *Electronics*, vol. 8, no. 2, p. 177, Feb. 2019. [Online]. Available: http://www.mdpi.com/2079-9292/8/2/177
[13] S. Quer, M. Andrea, and S. Giovanni, "The maximum common subgraph problem: A parallel and multi-engine approach," *Computation*, vol. 8, no. 2, pp. 1–29, 2020.
[14] J. Cohen. (2011). *Proof of Optimality of Minmax PIS Algorithm*. [Online]. Available: https://jcohen.name/papers/Cohen_minmax_2011.pdf
[15] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens, "Graph coloring on the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2019, pp. 231–240.
[16] Y. Chen, B. Brock, S. Porumbescu, A. Buluc, K. Yelick, and J. Owens, "Atos: A task-parallel GPU scheduler for graph analytics," in *Proc. 51st Int. Conf. Parallel Process.* New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 1–11, doi: 10.1145/3545008.3545056.
[17] Nvidia Corporation. *Cusparse Library Documentation*. Accessed: May 21, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html
[18] M. Osama, S. D. Porumbescu, and J. D. Owens, "Essentials of parallel graph analytics," in *Proc. Workshop Graphs, Architectures, Program., Learn. (GrAPL)*, May 2022, pp. 314–317. [Online]. Available: https://escholarship.org/uc/item/2p19z28q
[19] M. Osama, "Private communication," Jun. 2022.
[20] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011, doi: 10.1145/2049662.2049663.
[21] C. Gomes. (Apr. 2022). *Graph Coloring and Its Generalizations*. [Online]. Available: https://mat.gsia.cmu.edu/COLOR02/
[22] B. Rozemberczki and R. Sarkar, "Twitch gamers: A dataset for evaluating proximity preserving and structural role-based node embeddings," 2021, *arXiv:2101.03091*.

ALESSANDRO BORIONE received the degree in computer engineering from Politecnico di Torino, in 2022. He started coding during High School, when he discovered a passion for problem-solving. His research interest includes algorithm optimization.



LORENZO CARDONE (Graduate Student Member, IEEE) received the degree in computer science from Politecnico di Torino, where he is currently pursuing the Ph.D. degree. His main research interests include software parallelization and optimization, and he has recently started working in the field of hardware testing.

ANDREA CALABRESE (Member, IEEE) received the degree in computer science from Politecnico di Torino, in 2020, where he is currently pursuing the Ph.D. degree. His research interests include software optimization and parallel algorithms.

STEFANO QUER (Member, IEEE) received the M.S. degree in electronic engineering from Politecnico di Torino, Turin, Italy, in 1991, and the Ph.D. degree in computer engineering from the Ministry of University and Scientific and Technological Research, Rome, in 1996. He has been a Visiting Faculty with the Department of Electronic Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA. He has been an Intern with the Advanced Technology Group, Synopsys Inc., Mountain View, CA, USA, and the Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA, USA. He has also been a Compaq Computer Corporation Consultant. He is currently a Professor with the Department of Control and Computer Engineering, Politecnico di Torino. His main research interests include systems and tools for CAD for VLSI, formal methods for hardware and software systems, and embedded systems. Other activities focus on developing sequential and concurrent algorithms and optimizing techniques to achieve acceptable solutions with limited resources.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement