Graph Coloring on the GPU

Muhammad Osama*, Minh Truong*, Carl Yang*†, Aydın Buluç† and John D. Owens*
*Dept. of Electrical & Computer Engr., University of California, Davis
†Computational Research Division, Lawrence Berkeley National Laboratory
Emails: {mosama, mstruong, ctcyang, jowens}@ucdavis.edu, {ctcyang, abuluc}@lbl.gov

***Abstract*—We design and implement parallel graph coloring algorithms on the GPU using two different abstractions—one data-centric (Gunrock), the other linear-algebra-based (GraphBLAS). We analyze the impact of variations of a baseline independent-set algorithm on quality and runtime. We study how optimizations such as hashing, avoiding atomics, and a max-min heuristic affect performance. Our Gunrock graph coloring implementation has a peak 2× speed-up, a geomean speed-up of 1.3× and produces 1.6× more colors over previous hardwired state-of-the-art implementations on real-world datasets. Our GraphBLAS implementation of Luby's algorithm produces 1.9× fewer colors than the previous state-of-the-art parallel implementation at the cost of 3× extra runtime, and 1.014× fewer colors than a greedy, sequential algorithm with a geomean speed-up of 2.6×.**

***Index Terms*—parallel, GPU, graph coloring, graph algorithms**

I. Introduction

A graph $G = (V, E)$ is comprised of a set of vertices V together with a set of edges E, where $E \subseteq V \times V$. Graph coloring $C : V \rightarrow N$ is a function that assigns a color to each vertex that satisfies $C(v) \neq C(u)\ \forall (v, u) \in E$. In other words, graph coloring assigns colors to vertices such that no vertex has the same color as a neighboring vertex. Graph coloring is particularly useful in parallelizing computations for graphs (or graph-like data structures) such as the deterministic scheduling of dynamic computations [1]. Graph coloring is critical for the register allocation problem in compiler optimization [2], preconditioners for sparse iterative linear systems [3], [4], exam timetable scheduling [5], Sudoku solving [6], regularizing sparse matrix-matrix products [7], and approximating sparse Jacobians and Hessians that arise during automatic differentiation [8], [9].

Given a coloring C, many computations over same-colored vertices can be completely data-parallel, and computations iterate over all colors to process all vertices. Consequently it is desirable to minimize the number of colors in a graph coloring. However, a graph coloring that minimizes the number of distinct colors is NP-hard, difficult to approximate, and challenging to parallelize. In practice, various heuristics are used, and different algorithms and implementations of graph coloring exhibit tradeoffs between computation time and number of colors in the computed graph coloring.

Our contributions in this paper are as follows:

1) We survey parallel graph coloring algorithms on the GPU, and investigate how different optimizations such as hashing, avoiding atomics, and a max-min heuristic impact runtime and number of colors.
2) We show how the independent-set-based algorithm using Luby's Monte Carlo heuristic maps to a data-centric and a linear-algebra-based graph framework on the GPU, which are Gunrock [10] and GraphBLAS [11] respectively.
3) We demonstrate a peak 2× speed-up and a geometric mean speed-up of 1.3× over the previous hardwired state-of-the-art implementation [12], using Gunrock's high-level, bulk-synchronous, data-centric implementation of a parallel graph coloring algorithm.
4) We are the first to design a parallel graph coloring algorithm that uses linear-algebra-based primitives based on the GraphBLAS API. The implementation yields a coloring with 1.9× and 5.0× fewer colors than the two state-of-the-art graph coloring implementations by Naumov et al. [12], and 1.014× fewer colors than the greedy sequential algorithm in 1.92× less time.

II. Background & Related Work

Given a graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. A graph is undirected if for all $v, u \in V : (v, u) \in E \iff (u, v) \in E$. Otherwise, it is directed. The set of neighboring vertices to vertex v is called its adjacency *adj*(v).

The classic sequential "greedy" graph coloring algorithm works by using some ordering of vertices. Then it colors each vertex in order by using the minimum color that does not appear in its neighbors. While there exists an ordering that leads to the optimal number of colors, the problem of finding such a perfect ordering is NP-hard. Fortunately, certain orderings (such as ordering the vertices by degree from largest to smallest) can be used to bound the maximum number of colors to no more than one more than the maximum degree of the graph.

Algorithm 1 The parallel graph coloring algorithm.

Input: Graph $G = (V, E)$
Output: Array C of colors for each $v \in V$.

```
1: procedure ParallelColor(A)
2:     U ← V
3:     while |U| ≥ 0 do
4:         Choose an independent set I from U in parallel
5:         Color the vertices in I and put in C
6:         U ← U − I
7:     end while
8:     return C
9: end procedure
```

This greedy algorithm cannot be easily parallelized. Instead, one approach to parallel graph coloring uses independent sets. Shown in Algorithm 1, an independent set is found in parallel every iteration, which is then colored. Research has then been focused around how such an independent set is found. Luby's parallel maximal independent set algorithm is one such heuristic [13]. The Monte Carlo heuristic Luby proposes is the following:

1) For each $v \in V$, generate a random number $p(v)$
2) v is added to the independent set I if and only if $p(v) > p(w)$ for all $w \in adj(v)$.

Two heuristics that follow are: (1) Luby's algorithm in which the independent set I is maximal, and (2) a generalized Luby's algorithm in which I does not have to be maximal.

Jones and Plassmann propose a parallel graph coloring algorithm for asynchronous distributed execution [14]. In their computation model, they assign a single vertex to each processor and communicate colors between neighboring vertices. Each vertex then colors itself using the minimum color available to it.

A. Multi-threaded CPU

For a shared memory computation model, Gebremedhin and Manne [15] propose a greedy algorithm. Their algorithm has 3 phases: optimistic (speculative) coloring, conflict detection, and conflict resolution. The first phase involves assigning a batch of vertices to different processors, assigning the minimum color available to a vertex (taking into account both local and remote neighbors), synchronizing with other processors, and repeating. In doing so, however, different processors may color two neighboring vertices in the same step. That motivates the conflict detection phase, which is done in parallel, and the conflict resolution phase, which is done sequentially. Deveci et al. [16] modify the Gebremedhin-Manne algorithm for Xeon Phi and GPU. Using the work-span model, Hasenplaugh et al. [17] study the various impact of different orderings on the parallel complexity, runtime and quality of ordering.

B. Distributed CPU

To the best of our knowledge, the first distributed-memory implementations of graph coloring algorithms are due to Allwright et al. [18]. They implemented Luby's algorithm, the Jones-Plassmann heuristic, as well as two greedy heuristics: smallest-degree-last and largest-degree-first, on both the CM-5 and the Intel iPSC/860 computers. They found that smallest-degree-last greedy heuristic used the fewest number of colors. In terms of performance, Jones-Plassmann and largest-degree-first were the two fastest codes.

Bozdağ et al. [19] extend the work of Gebremedhin and Manne to a graph coloring framework in distributed memory. Their scheme also involves speculative coloring, conflict detection, and finally conflict resolution. The advantage of greedy algorithms compared to independent-set-based algorithms such as Jones-Plassmann is that they often result in fewer colors. More recently, Salihoğlu and Widom [20] implement a similar scheme on Pregel-like systems where they apply various additional optimizations such as finishing computations serially.

C. Handcoded GPU

The first graph coloring work on the GPU was done by Grosset et al. [21]. Their algorithm is based on the Gebremedhin-Manne algorithm, and they find that they can color with fewer colors than distributed graph coloring implementations. Naumov et al. [12] implement a state-of-the-art implementation csrcolor using the popular cuSPARSE library. Their algorithm implements the generalized Luby's algorithm. Che et al. [22] study variations of the Jones-Plassmann algorithm. They observe a static work allocation runs into load-imbalance problems, so they use a largest degree-first strategy for early iterations, followed by a randomized strategy.

III. Graph Processing Frameworks

We consider two graph processing frameworks for the GPU: GraphBLAS [11] and Gunrock [10].

A. GraphBLAS

Several independent systems use matrix algebra to perform graph operations [23]–[25]. GraphBLAS is an effort by the graph analytics community to unify such efforts into a single, unified API [26], [27]. The goal of the GraphBLAS API specification is to outline the common, high-level operations—such as vector-vector inner product, matrix-vector product, and matrix-matrix product—and define the standard interface for scientists to use these functions in a hardware-agnostic manner. This way, the runtime of the GraphBLAS implementation can make the difficult decisions about optimizing each of the GraphBLAS operations on a given piece of hardware. In this paper, instead of defining our own functions, we use the functions from the GraphBLAS API when we describe our algorithms.

We give an informal introduction to the five GraphBLAS operations that we use in the GraphBLAS-based graph coloring algorithm (Algorithm 2), but an interested reader can consult the API document for more details [26].

1) Assign to a vector (GrB_assign): Assigns a scalar to a vector using a mask, which is a core concept of GraphBLAS. The mask controls whether a result of the computation will be written to the output array. As an example, let us consider the elementwise multiplication operation for vectors a, b, c using a mask vector m i.e. $c \leftarrow a \times b \,.*\, m$. If the mask element $m[i]$ is C-style castable to 0, then computation result $c[i]$ is unchanged. However, if the mask element is C-style castable to 1, then the computation result $c[i] = a[i] \times b[i]$. If we take $=$ to mean "C-style castable", then:

$$c[i] \leftarrow \begin{cases} c[i], & \text{if } m[i] = 0 \\ a[i] \times b[i], & \text{if } m[i] = 1 \end{cases}$$

At a high-level, masking has proved to be important for performance, because we can avoid many memory accesses when the mask is 0 [28].

Algorithm 2 Parallel independent set graph coloring algorithm implemented in linear algebra (GraphBLAS).

Input: Adjacency matrix A of graph $G = (V, E)$, already built empty vectors *C*, *weight*, *frontier*
Output: Array C of colors for each $v \in V$.

```
1:  procedure GraphblasColor(A, C)
2:      ▷ Initialize colors to 0
3:      GrB_assign(C, GrB_NULL, 0, GrB_ALL, nrows(A), desc);
4:      ▷ Assign random weight to each vertex
5:      GrB_apply(weight, GrB_NULL, GrB_NULL,
            set_random(), weight, desc);
6:      for each color = 1, ..., n do
7:          ▷ Find max of neighbors
8:          GrB_vxm(max, GrB_NULL, GrB_NULL,
                GrB_INT32MaxTimes, weight, A, desc);
9:          GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL,
                GrB_INT32GT, weight, max, desc);
10:         ▷ Find all largest nodes that are uncolored
11:         GrB_reduce(succ, GrB_NULL, GrB_INT32Plus,
                frontier, desc);
12:         ▷ Stop when frontier is empty
13:         if succ = 0 then
14:             break;
15:         end if
16:         ▷ Assign new color
17:         GrB_assign(C, frontier, GrB_NULL, color, GrB_ALL,
                nrows(A), desc);
18:         ▷ Get rid of colored nodes in candidate list
19:         GrB_assign(weight, frontier, GrB_NULL, 0,
                GrB_ALL, nrows(A), desc);
20:     end for
21: end procedure
```

2) Apply user-defined function (GrB_apply): Applies the user-defined function to each element of a vector. In this case, we are using 'set_random()' to set each vector element to a random integer.

3) Vector-matrix multiply (GrB_vxm): Multiplies a vector by a matrix. The GraphBLAS API hides the distinction between sparse vs. dense vectors and matrices from the user, but instead allows the implementation to internally call different subroutines based on input sparsity. One core concept of GraphBLAS is that it relies on overloading the standard arithmetic semiring through the concept of generalized semirings. For example, the matrix-vector multiplication is done using the max-times semiring 'GrB_INT32MaxTimes' $(\max, \times, \mathbb{R})$, which overloads the standard arithmetic semiring $(\times, +, \mathbb{R})$. Our implementation uses a proposed addition [29] to the standard GraphBLAS API called predefined semirings, which avoids the user having to use the 'GrB_Semiring' interface to build the matrix-multiplication operation they want.

4) Elementwise add (GrB_eWiseAdd): Elementwise adds a vector with another vector.

5) Vector reduction (GrB_reduce): Reduces a vector to a scalar.

B. Gunrock

Gunrock is a parallel graph analytics library that employs a high-level data-centric abstraction focused on operations on vertex or edge frontiers [10]. Hidden from the programmer, Gunrock integrates sophisticated load-balancing and work-efficiency strategies into its core. These strategies are exposed to the programmer using a high-level API as Gunrock's operators. In this paper we will leverage the following Gunrock's high-performance operators to express our algorithms, and measure and compare the performance of the different implementations:

1) Advance Operator: An advance operator is used to generate a new frontier from the current frontier by visiting the neighbors of the current frontier. Each input item maps to multiple output items from the input item's neighbor list.

2) Compute Operator: A compute operator defines an operation on all elements (vertices or edges) in its input frontier. A programmer-specified compute operator can be used together with a traversal operator such as advance. Gunrock performs that operation in parallel across all elements without regard to order.

3) Neighbor-Reduce Operator: A neighbor-reduce operator uses the advance operator to visit the neighbor list of each item in the input frontier and performs a segmented reduction over the neighborhood (neighbor list) generated via the advance.

IV. Implementation Mapping to Frameworks

We implemented our parallel graph coloring algorithms using two GPU graph processing frameworks: GraphBLAS and Gunrock. It is challenging to write hardwired graph algorithms on the GPU, so our goal is to find out whether these two frameworks are flexible enough to design and implement a graph coloring algorithm, and whether the result will be performance-competitive with the state of the art.

In both frameworks, we input compressed sparse row (CSR) sparse matrix format, which is commonly used for graph analytics. In CSR, one array stores a list of neighbor nodes and another array stores the offset of the neighbor list for each node. The column-indices array and row-offsets array are equivalent to the neighbor nodes list and the offset list in the basic adjacency list definition.

A. GraphBLAS

1) Independent Set Graph Coloring: Algorithm 2 shows the Independent Set (IS) graph coloring algorithm designed using the GraphBLAS API. This will be the base algorithm, which we will modify to do maximal independent set and Jones-Plassman graph coloring. We begin at Line 3 by initializing the color vector C to 0, which means each vertex is uncolored. This is done using the GrB_assign function. Next, each vertex of the *weight* vector must be initialized to a random integer using a user-defined function 'set_random()'.

For each vertex v of the *weight* vector, we find the max weighted neighbor in $adj(v)$. This operation is performed using GrB_vxm in Line 8 of Algorithm 2 where we multiply the *weight* vector with adjacency matrix A on the $(\max, \times, \mathbb{R})$ semiring. Next, to find the independent set, we compare the max weighted neighbor of each vertex with its own weight using the GrB_eWiseAdd function with the GrB_INT32GT binary function. This function compares two integer values and returns true if the lefthand side element is greater than

the righthand side element, and false otherwise. The output in the *frontier* vector will be the independent set this iteration. Next, we compute a reduction using GrB_reduce to determine the size of the independent set. If it is zero, then we are done. Otherwise, we must color it and eliminate it from the candidate list *weight*. These two operations can be done with GrB_assign.

Algorithm 3 Maximal independent set graph coloring inner loop implemented in linear algebra (GraphBLAS).

Input: Adjacency matrix A of graph $G = (V, E)$, already built empty vectors *mis*, *weight*, *frontier*
Output: Maximal independent set vector mis.

```
1:  procedure GRAPHBLASMISINNER(A, mis)
2:      ▷ Initialize MIS array to 0
3:      GrB_assign(mis, GrB_NULL, 0, GrB_ALL, nrows(A),
            desc);
4:      do
5:          ▷ Find max of neighbors
6:          GrB_vxm(max, weight, GrB_NULL,
                GrB_INT32MaxTimes, weight, A, desc);
7:          ▷ Find all largest nodes are candidates
8:          GrB_eWiseAdd(frontier, GrB_NULL, GrB_NULL,
                GrB_INT32GT, weight, max, desc);
9:          ▷ Assign new members (frontier) to independent set
10:         GrB_assign(v, f, GrB_NULL, 1, GrB_ALL, nrows(A),
                desc);
11:         ▷ Eliminate frontier from candidate list
12:         GrB_assign(weight, frontier, GrB_NULL, 0, GrB_ALL,
                nrows(A), desc);
13:         ▷ Stop when frontier is empty
14:         GrB_reduce(succ, GrB_NULL, GrB_INT32Plus,
                frontier, desc);
15:         if succ = 0 then
16:             break;
17:         end if
18:         ▷ Remove neighbors of frontier from candidates
19:         GrB_vxm(max, weight, GrB_NULL, GrB_Boolean,
                frontier, A, desc);
20:         GrB_assign(weight, max, GrB_NULL, 0, GrB_ALL,
                nrows(A), desc);
21:     while succ > 0
22: end procedure
```

2) Maximal Independent Set Graph Coloring: To implement maximal independent set graph coloring, we replaced Lines 8 and 9 of Algorithm 2 with a call to GRAPHBLASMISINNER (Algorithm 3). The main difference is that instead of using the Monte Carlo heuristic proposed by Luby once, we keep adding vertices to the independent set until it is maximal. Only when it is maximal do we color the independent set. Therefore, the main difference is that we introduce a do-while loop on Line 4. In order to add vertices to the independent set, the problem of conflicts in the next iteration must be solved. This is done by doing a second traversal per iteration in order to find the independent set's neighbors, which can then be removed. These two operations are done by the GrB_vxm and GrB_assign on Lines 19 and 20.

3) Jones-Plassman Graph Coloring: To implement Jones-Plassman graph coloring, we replaced Lines 8 and 9 of Algorithm 2 with a call to GRAPHBLASJPINNER (Algorithm 4). The primary challenge that needs to be addressed in Jones-Plassman is after determining the candidate independent set *frontier*, finding the minimum color available to all these vertices.

Algorithm 4 Parallel Jones-Plassman graph coloring algorithm helper function implemented in linear algebra (GraphBLAS).

Input: Adjacency matrix A of graph $G = (V, E)$, color vector C, random vector *weight*, independent set *frontier*, already built empty vector *colors*, vector *ascending* that has been filled with numbers 0, 1, 2, ... *max_colors*
Output: Minimum available color min_color.

```
1:  procedure GRAPHBLASJPINNER(A, C)
2:      ▷ Find neighbors of frontier
3:      GrB_vxm(max, C, GrB_NULL, GrB_Boolean, frontier,
            A, desc);
4:      ▷ Get min color
5:      GrB_eWiseMult(n, GrB_NULL, GrB_NULL,
            GrB_INT32PlusMul, max, C, desc);
6:      ▷ Fill possible colors array
7:      GrB_assign(colors, GrB_NULL, GrB_NULL, 0,
            GrB_ALL, nrows(A), desc);
8:      ▷ Scatter nodes into possible colors array
9:      GxB_scatter(colors, GrB_NULL, n, max_colors, desc);
10:     ▷ Map boolean array to element id
11:     GrB_eWiseMult(min_array, GrB_NULL, GrB_NULL,
            GrB_INT32MinPlus, colors, ascending, desc);
12:     GrB_Vector_setElement(min_array, max_colors, 0);
13:     ▷ Compute min color
14:     GrB_reduce(min_color, GrB_NULL, GrB_INT32Max,
            min_array, desc);
15:     return min_color;
16: end procedure
```

The latter task can be formulated as follows: We have a set of colors represented by the natural numbers and wish to find the smallest number not in the set. We implement this with a scatter (Line 9) to an array of possible colors *colors*:

$$\text{colors[n[i]]} = \text{max\_colors[i]} \text{ for all i} \in \text{n}$$

However, this scatter could not be done within the confines of the GraphBLAS API. Therefore, we needed a GraphBLAS extension operation GxB_scatter. Next, the first zero in this Boolean array must be found. This can be done by comparing the Boolean array to an ascending integer array, and returning 1 if the two array's values match and a 0 if they do not. Finally, a min-reduction on the Boolean array yields the minimum available color.

B. Gunrock

1) Independent Set Graph Coloring: The Independent Set (IS) graph coloring algorithm finds an independent set of vertices to be colored based on random-number comparisons. Neighboring vertices compare their pre-assigned random numbers with one another. The independent color set contains only those vertices that possess the largest random numbers relative to their neighbors. Every vertex inside the independent color set can then be painted with the same color because they are guaranteed to not be neighbors.

Our Gunrock implementation of this algorithm follows. In it, a compute operator inputs a frontier of all vertices and

compares each vertex's random number to its neighbors' in parallel. A for loop within each thread execution flow checks the vertex's assigned random number with its neighbor's serially on Lines 25–35 of Algorithm 5. This results in load imbalance because each vertex has a different degree. Thread divergence is also a concern, because the random number comparison divides the frontier into vertices that belong to the independent color set and vertices that do not. The Gunrock enactor iteratively calls this compute operator until all vertices are colored on Line 9 of Algorithm 5. The algorithm checks for valid vertices' colors by invoking another compute operator after every coloring. The thread execution, when checking for valid colors, atomically counts the number of vertices in the independent color set. If the count is equal to the number of vertices in the graph, then all vertices in the graph have been successfully colored and Gunrock can stop the executing iteration loop.

An optimization for this implementation involves forming two independent sets every iteration. Instead of only assigning vertices with the largest random number relative to their neighbors to a maximum independent color set, the implementation also assigns colors to vertices with the *smallest* random number to a minimum independent color set. Because the max-comparison and min-comparison sets must be mutually exclusive, we can perform assignment on two colors every iteration with no additional overhead, amortizing the cost of the serial for loop (Lines 33 and 41 of Algorithm 5). This optimization reduces the coloring time almost by half.

2) Hash and Independent Set Graph Coloring: We propose a Hash Independent Set algorithm, which is a modification of the Independent Set algorithm. Each vertex in the frontier compares only its neighbors with one another, and adds the neighbor vertex with the largest random number relative to all neighbors to the color set. This method guarantees one color proposal per vertex, because for every vertex there exists at least one neighbor vertex with the largest random number. This means the Hash IS color set can contain more vertices than the independent color set of min-max IS (Lines 20–24 of Algorithm 6). The color set is not an independent set, unlike the IS color proposal, because each vertex knows only its local topology. Asserting that one of the neighbors can be colored without knowing the neighbor's connections can result in a color conflict, since neighboring vertices can be added to the color set by different proposing vertices. In min-max IS, a vertex either forfeits its access to the independent color set (if a neighbor has a better random number), or adds itself to the set only when all neighbor vertices forfeit their accesses. Having a larger color set means Hash IS solutions have fewer iterations and a potentially fewer number of colors. It also means the coloring process is not an exact solution, and needs a conflict resolution scheme.

The conflict resolution is another compute operation. It checks all colored vertices with their neighbors in a serial for loop similar to the min-max IS coloring scheme. If the resolution detects a color conflict, it resets one of the violating vertices to uncolored. In general, the implementation sacrifices

Algorithm 5 Parallel graph coloring algorithm implemented in Gunrock with min-max coloring optimization.

Input: Frontier $F = (V)$ for all $V \in G$.
Output: Array C of colors for each $v \in V$.

```
1:  ▷ Gunrock's color primitive driver.
2:  procedure GunrockColor(F, C, R)
3:      Initialize iteration ← 0
4:      ▷ Initialize colors to be invalid
5:      Initialize C ← c ∀c = invalidColor
6:      ▷ Assign random weight to each vertex
7:      Initialize R ← generateRandomNumbers
8:      Initialize F ← v ∀v ∈ G
9:      while ∀c ∈ C is not valid do
10:         ▷ Call coloring compute operator using a parallel forall
11:         F ← ComputeOp(
          ColorOp(iteration, C, R), F)
12:     end while
13: end procedure
14: ▷ Gunrock's Compute Coloring operator
15: procedure ColorOp(iteration, C, R)
16:     v = F[threadIdx]
17:     ▷ If already colored, return
18:     if C[v] is valid then:
19:         return
20:     end if
21:     Initialize colormax ← true
22:     Initialize colormin ← true
23:     color ← 2 * iteration
24:     ▷ Visit all neighbors of an active node and find the minimum
      and maximum random number
25:     for u ∈ Neighbor(v) do
26:         if C[u] is valid
          &  C[u] ≠ color + 1
          &  C[u] ≠ color + 2 then
27:             continue
28:         end if
29:         if R[v] ≤ R[u] then
30:             colormax ← false
31:         end if
32:         if R[v] ≥ R[u] then
33:             colormin ← false
34:         end if
35:     end for
36:     ▷ If active vertex is the maximum or minimum, color it
37:     if colormax then
38:         C[v] ← color + 1
39:     end if
40:     if colormin then
41:         C[v] ← color + 2
42:     end if
43: end procedure
```

fast runtime for fewer colors. To amortize the cost of the conflict resolution, the implementation uses a hash table to inform the vertex about previous colors that cannot be used. Based on this partial knowledge, the vertex can choose to either use previous colors that are not prohibited by the hash table or use a new color generated every iteration. Doing so potentially reduces the total number of colors used. Because the hash table does not store all prohibited colors for a vertex, the vertex can end up using one of those colors. This means the conflict resolution scheme can check for conflicts due to non-independent color set and the use of prohibited colors at

the same time. Empirically, using the hash table can reduce the total number of colors by 1 or 2. Our hash table reserves a fixed number of entries per vertex. A hash generation operator populates colors that vertices cannot use based on the vertices' neighbor. The hash table size is a modifiable value, and is inversely related to the number of conflicts because the table does not guarantee storing all prohibited colors. A Gunrock compute operator automatically updates the table every coloring iteration after new vertices are colored. If all entries for a vertex are filled, the table ignores new colors for that vertex. Better hash functions can replace the random number comparison with a color proposal based on color sets of previous iterations.

Algorithm 6 Parallel graph coloring algorithm implemented in Gunrock with hash coloring optimization.

Input: Frontier $F = (V)$ for all $V \in G$.
Output: Array C of colors for each $v \in V$.

1: ▷ Gunrock's Compute Hash Coloring operator
2: **procedure** HASHCOLOROP(**iteration**, **C**, **R**, **H**)
3: $v = F[threadIdx]$
4: ▷ If already colored, return
5: **if** $C[v]$ is valid **then**:
6: **return**
7: **end if**
8: Initialize $max/min \leftarrow v$
9: Initialize $temp \leftarrow R[v]$
10: $color \leftarrow 2 * iteration$
11: **for** $u \in Neighbor(v)$ **do**
12: **if** $R[u] > temp$ & $C[u]$ is not valid **then**
13: $max \leftarrow u$
14: **end if**
15: **if** $R[u] < temp$ & $C[u]$ is not valid **then**
16: $min \leftarrow u$
17: **end if**
18: **end for**
19: ▷ Reuse existing color first if possible
20: **for** $c \in UsedColors$ **do**
21: **if** $c \notin H(max/min, u) \forall u \in Neighbor(max/min)$ & $C[max/min]$ is not valid **then**
22: $C[max/min] \leftarrow c$
23: **end if**
24: **end for**
25: ▷ If existing colors result in conflict, use new color
26: **if** $C[max/min]$ is not valid **then**
27: $C[max/min] \leftarrow color + 1$ / $color + 2$
28: **end if**
29: **end procedure**

3) Advance Neighbor-Reduce Graph Coloring: The Advance Neighbor-Reduce coloring implementation eliminates the serial for loop found inside the min-max IS operator on Lines 25–35 of Algorithm 5 with a parallel reduce on Lines 24–28 of Algorithm 7. This implementation uses Gunrock's Advance operator to gain access to all neighboring vertices in parallel. The Reduce operator then compares all vertex-assigned random numbers to all neighbor vertices in parallel, flagging to-be-colored vertices. A compute operator then colors all flagged vertices in parallel.

This implementation is similar to min-max IS because Advance Reduce operators return an independent color set. However, because the Reduce operator can only perform binary operations (either max or min comparison), the implementation cannot paint two colors per iteration. This is because the Reduce operator consumes the Advance neighbor frontier; reusing the frontier for a second comparison is not permitted without launching another neighbor-reduce operation (one for max reduction, one for min reduction). Another future optimization is to fuse the max and min operations and use a single reduce operator to avoid a global synchronization.

Algorithm 7 Parallel graph coloring algorithm implemented in Gunrock using Advance Reduce operator.

Input: Frontier $F = (V)$ for all $V \in G$.
Output: Array C of colors for each $v \in V$.

1: ▷ Gunrock's Advance-Reduce color primitive driver.
2: **procedure** GUNROCKARCOLOR(**F**, **C**, **R**)
3: Initialize $iteration \leftarrow 0$
4: Initialize $C \leftarrow c \; \forall c = invalidColor$
5: Initialize $R \leftarrow generateRandomNumbers$
6: Initialize $F \leftarrow v \; \forall v \in G$
7: Initialize $Removed \leftarrow []$
8: **while** $\forall c \in C$ is not valid **do**
9: ▷ Create a neighborhood frontier and reduce
10: $F \leftarrow NeighborReduceOp($ $AdvanceOp(F, Removed),$ $ReduceMaxOp(Removed))$
11: $F \leftarrow ComputeOp($ $ColorRemovedOp(iteration, C, Removed))$
12: **end while**
13: **end procedure**
14: ▷ Advance operator: Visit all neighbors to reduce
15: **procedure** ADVANCEOP(**F**, **Removed**)
16: $v = F[blockIdx]$
17: **if** $Neighbor(v, threadIdx) \notin Removed$ **then**
18: **return** $Neighbor(v, threadIdx)$
19: **end if**
20: **end procedure**
21: ▷ Reduce a neighbor segment based on the max random number
22: **procedure** REDUCEMAXOP(**Removed**)
23: $a = F[2 * threadIdx + 1]$
24: $b = F[2 * threadIdx + 2]$
25: $Reduced.append((a < b) \; ? \; b \; : \; a)$
26: **end procedure**
27: ▷ Color vertices that are removed from frontier
28: **procedure** COLORREMOVEDOP(**iteration**, **C**, **Removed**)
29: $v = Removed[threadIdx]$
30: **if** $C[v]$ is valid **then**:
31: **return**
32: **end if**
33: $C[v] = iteration$
34: **end procedure**

V. EXPERIMENTS & DISCUSSION

A. Experimental setup

We ran all experiments in this paper on a Linux workstation with 2× 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 556 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA's nvcc compiler (version 9.1.85). The C code was compiled using gcc 5.4.0. All graph coloring tests were run 10 times with the average runtime used for

Dataset	Vertices	Edges	Avg. Degree	Diameter	Type
offshore	260k	4.2M	17.33	41*	ru
af_shell3	505k	17.6M	35.84	485*	ru
parabolic_fem	1.1M	112.8M	8	1536*	ru
apache2	7.4M	4.8M	7.74	449*	ru
ecology2	1M	5M	6	1998*	ru
thermal2	4.2M	483M	8	1778*	ru
G3_circuit	1.6M	7.7M	5.83	515*	ru
FEM_3D_thermal2	148k	3.5M	24.6	150	rd
thermomech_DK	204k	2.8M	14.93	647*	rd
ASIC_320ks	322k	1.3M	6.68	45	rd
cage13	445k	7.5M	17.8	42*	rd
atmosmodd	1.3M	8.8M	7.94	351*	rd
rgg_n_2_15_s0	32.8k	320k	9.78	191	gu
rgg_n_2_16_s0	65.6k	684k	10.44	254	gu
rgg_n_2_17_s0	131k	1.5M	11.11	341	gu
rgg_n_2_18_s0	262k	3.1M	11.8	464	gu
rgg_n_2_19_s0	524k	6.5M	12.47	632*	gu
rgg_n_2_20_s0	1M	13.8M	13.14	865*	gu
rgg_n_2_21_s0	2.1M	29M	13.81	1182*	gu
rgg_n_2_22_s0	4.2M	60.7M	14.47	1621*	gu
rgg_n_2_23_s0	8.4M	127M	15.14	2230*	gu
rgg_n_2_24_s0	16.8M	265.1M	15.8	2622	gu

TABLE I: Dataset Description Table. Graph types are: r: real-world, g: generated, u: undirected, d: directed. An asterisk (*) indicates diameter is an estimate using samples from 10,000 vertices.

results. The implementation of GraphBLAS we use is called GraphBLAST [11].

The datasets we used are listed in Table I. The rgg randomized graphs were generated for DIMACS10 [30], [31], and all other graphs are from the SuiteSparse Matrix Collection [32]. All datasets have been converted to undirected graphs, and self-loops and duplicated edges are removed.

B. Performance Summary of Gunrock

Figure 1a compares runtime for our Gunrock and GraphBLAST implementations and state-of-the-art Naumov et al. JPL and CC GPU implementations with a baseline CPU implementation. In general, our Gunrock's Independent Set implementation shows better performance over Naumov et al.'s JPL implementation with a comparable color count because of our min-max independent set optimization, essentially generating two independent sets for every iteration. Gunrock's IS implementation uses a compute operator (not load-balanced), which as the results show outperforms Gunrock's AR implementation (load-balanced), because the overhead of doing complex load-balancing when using advance and segmented reduction on neighbors is more taxing than simply assigning each active thread to a vertex and generating two independent sets per iteration than one. Gunrock's Hash implementation doesn't perform well against Naumov et al.'s JPL implementation due to the global synchronization required after generating the sets before resolving any color conflicts. The conflict resolution operator and the hash generation within the hash implementation also contribute to the slowdown when compared against Gunrock's IS implementation.

Optimization	Performance (ms)	Speedup
Baseline (Advance-Reduce)	656	—
Hash Color	17.21	38.11×
Independent Set with Atomics	13.67	1.26×
Independent Set without Atomics	11.15	1.23×
Min-Max Independent Set	6.68	1.67×

TABLE II: Impact of Gunrock's optimizations on the performance measured in elapsed time (ms) on the G3_circuit dataset with approximately 1.6M vertices and 7.7M edges.

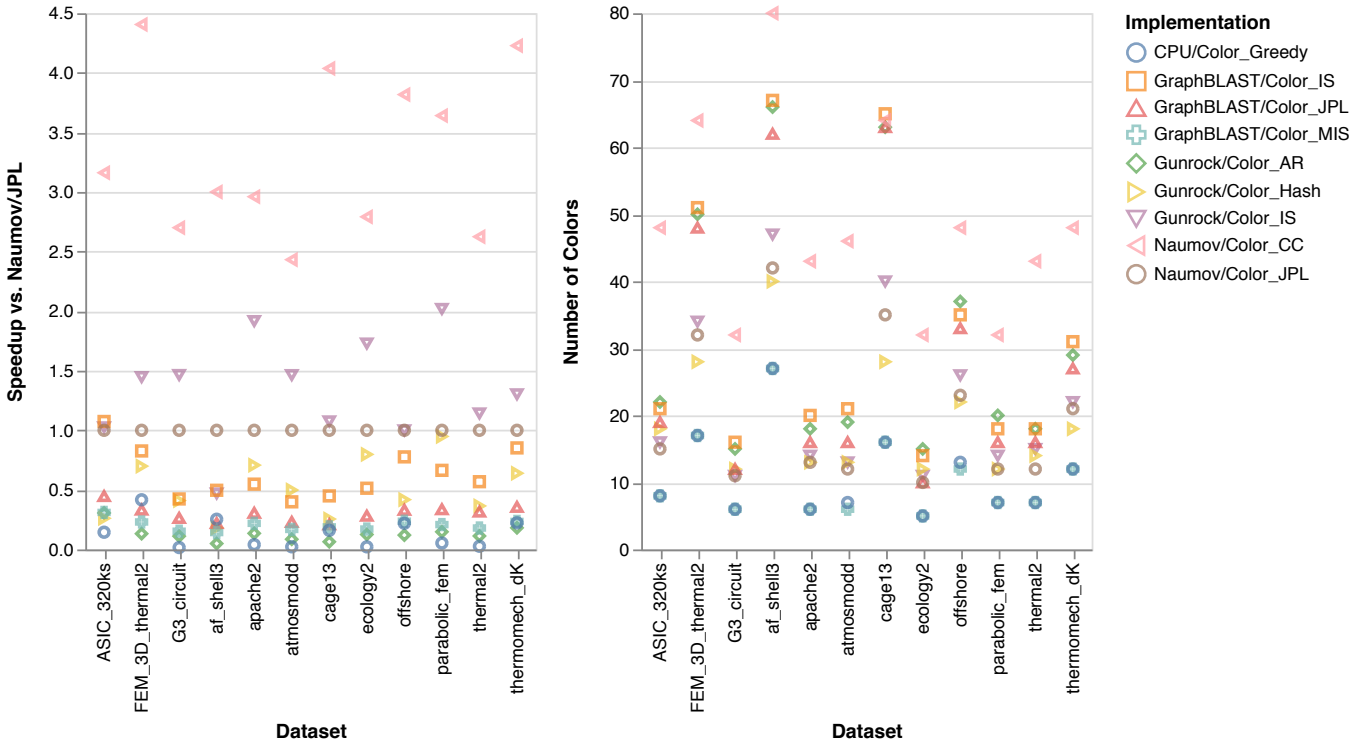
Our Independent-Set-based graph coloring achieves a peak performance of $2\times$ on the *parabolic_fem* dataset, and a geometric mean of $1.3\times$ compared to Naumov's JPL implementation, while maintaining a comparable color count. The performance gain observed against Naumov's implementation is mainly due to the two independent set coloring per iteration optimization. Gunrock's IS implementation also avoids atomics and global synchronization unlike other Gunrock implementations. The flaw of the IS implementation is the serial for-loop within the compute operator that visits all neighbors per active vertex. The performance degradation due to the serial for-loop is clearly visible in the *af_shell3* dataset (as shown in Figure 1a, a slowdown of $0.47\times$ compared to Naumov's JPL implementation), where the average degree of the graph is 35.84, much higher than some of the other test datasets (see Table I).

Our Hash-based graph coloring expands on IS min-max coloring by using two additional compute operators for conflict resolution and hash generation. Due to the additional operators we are able to reuse colors for every iteration and generate a lower color count than the IS implementation (see Figure 1b). Due to the additional operators, we now require two global synchronizations, one after each operator, causing the slowdown when compared to the IS implementation.

Our Advance Neighbor-Reduce based graph coloring performs poorly against the state-of-the art and other Gunrock implementations. The goal of this implementation was to eliminate the serial for-loop within the Gunrock's IS and Hash implementation with a parallel segmented reduce. However, the overhead of performing a load-balanced advance-segmented reduce with two global synchronizations is more than simply assigning each vertex to an active thread with a serial for-loop. The bottleneck of the AR implementation is the segmented reduction operation within the neighbor-reduce, internally performed by assigning segments to threads, warps or blocks depending on the size of the segment.

C. Performance Summary of GraphBLAST

On the real-world data shown in Figure 1a, the three GraphBLAST implementations in terms of runtime can be listed from slowest to fastest as independent set, Jones-Plassman, and maximal independent set. The latter two are $1.98\times$ and $3\times$ slower than the independent set baseline. The fastest out of the three implementations is slower than Naumov by $1.66\times$. In terms of number of colors as shown in Figure 1b,



(a) Runtime Comparison

(b) Number of Colors Comparison

Fig. 1: Speedup and color count comparison of our implementations on various datasets vs. Naumov et al. implementations and greedy CPU implementation.

the order of best to worst reverses: maximal independent set, Jones-Plassman and independent set. This is evidence of the time-quality tradeoff often seen across graph coloring algorithms [9]. The latter two need 2.5× and 2.9× more colors than maximal independent set. Compared to Naumov, 1.9× fewer colors are used. Compared to the greedy sequential implementation, maximal independent set yields 1.02× fewer colors using 2.6× less time.

To explain the differences in runtime, we ran some profiling of GPU kernels. We find that for maximal independent set and Jones-Plassman as compared to the independent set, a second call to GrB_vxm ends up taking nearly 50% of the runtime. For Jones-Plassman in particular, the call on Line 7 can be optimized by using GrB_assign rather than using a cudaMemcpyHostToDevice operation. For maximal independent set, the inner loop needs to run potentially for many iterations, which causes the runtime to increase.

D. Time-quality Trade-off

Figure 2 shows the time-quality tradeoff between different Gunrock implementations and between different GraphBLAST implementations. In the case of Gunrock, using a more compute-intensive implementation such as hashing the color leads to fewer colors used than independent set. Similarly for GraphBLAST, using maximal independent set, which as noted above takes a second call to GrB_vxm and an inner loop to repeat traversals until the independent set is maximal, uses more time to converge. However once a coloring is found, it is higher quality than one found using the independent set algorithm.

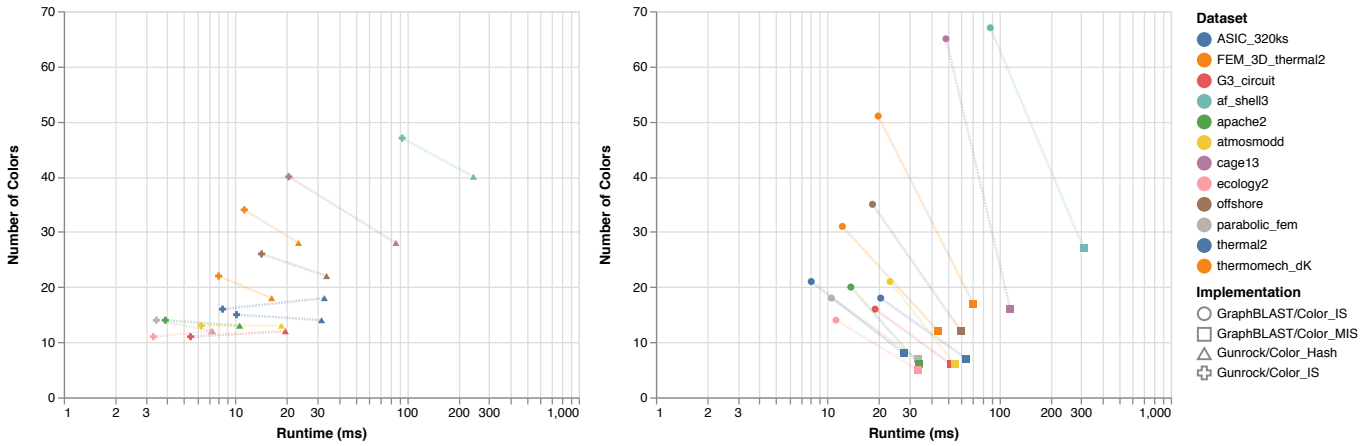
E. Scalability Summary using Randomly Generated Graphs (RGG)

On the synthetic data used for scaling as shown in Figure 3b, we show the best performers in terms of runtime from GraphBLAST and Gunrock, which are independent set implementations in both cases. We see that Gunrock does better for smaller graphs, which indicates that it has lower overhead. GraphBLAS begins to do better beyond scale 23 and 24. In terms of numbers of colors as shown in Figure 3d, the Gunrock implementation requires 1.14× fewer colors.

When comparing colors in Figure 1b, we show that our GraphBLAST MIS implementation outperforms all other implementations, even generating better color count than a greedy CPU algorithm. Gunrock's Hash implementations also show promising results in the number of generated colors, as they are able to better utilize colors while resolving conflicts, and reuse some of the colors assigned in the past iterations. Gunrock's IS and AR implementations also generate comparable color counts to Naumov's JPL implementation.

VI. Conclusions

In this paper, we designed and implemented parallel graph coloring algorithms on the GPU using two different abstrac-



(a) Number of Colors vs. Runtime for two Gunrock implementations

(b) Number of Colors vs. Runtime for two GraphBLAST implementations

Fig. 2: Different implementations offer different tradeoffs between runtime and color count; for both Gunrock and GraphBLAST, we can generally use a more expensive implementation and achieve better color counts.

tions: one data-centric (Gunrock), the other linear-algebra-based (GraphBLAS). We analyzed the impact of variations of a baseline independent set algorithm on quality and runtime. We examined how optimizations such as hashing, avoiding atomics and a max-min heuristic affects performance. We demonstrated our Gunrock graph coloring implementation has a peak 2× speed-up and geomean speed-up of 1.3× over a previous hardwired state-of-the-art implementation that produces a coloring of similar quality. We showed our GraphBLAS implementation of Luby's algorithm produces 1.9× fewer colors than the previous state-of-the-art parallel implementation and 1.014× fewer colors than a greedy, sequential algorithm.

One limitation of this work is that it focuses on comparisons of different variants of either Jones-Plassman or Luby's Monte Carlo heuristic. A possible future research direction would be to compare these algorithms with Gebremedhin-Manne on the GPU. Another future research direction would be to examine how the largest-degree-first heuristic compares with the randomized algorithms we used. In this work, we primarily looked at mesh graphs. With power law graphs, is possible that a random weight initialization would perform worse than largest-degree first, because random weight initialization will make it more likely a node with few neighbors is colored rather than a node with many neighbors being colored as in the case of largest-degree-first.

Acknowledgments

We would like to thank Maxim Naumov for explaining technical details about his implementation. We appreciate the funding support from the Defense Advanced Research Projects Agency (Awards # FA8650-18-2-7835 and HR0011-18-3-0007) and the National Science Foundation (Awards # OAC-1740333 and CCF-1629657). This work is also supported in part by the Applied Mathematics program of the DOE Office of Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231, and in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson, "Executing dynamic data-graph computations deterministically using chromatic scheduling," *ACM Transactions on Parallel Computing*, vol. 3, no. 1, p. 2, 2016.

[2] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[3] M. T. Jones and P. E. Plassmann, "Scalable iterative solution of sparse linear systems," *Parallel Computing*, vol. 20, no. 5, pp. 753–773, 1994.

[4] Y. Saad, "ILUM: a multi-elimination ILU preconditioner for general sparse matrices," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 830–847, 1996.

[5] F. T. Leighton, "A graph coloring algorithm for large scheduling problems," *Journal of Research of the National Bureau of Standards*, vol. 84, no. 6, pp. 489–506, 1979.

[6] F. Akman, "Partial chromatic polynomials and diagonally distinct Sudoku squares," *arXiv preprint arXiv:0804.0284*, 2008.

[7] M. McCourt, B. Smith, and H. Zhang, "Sparse matrix-matrix products executed through coloring," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 90–109, 2015.

[8] T. F. Coleman and J. J. Moré, "Estimation of sparse Hessian matrices and graph coloring problems," *Mathematical Programming*, vol. 28, no. 3, pp. 243–270, 1984.

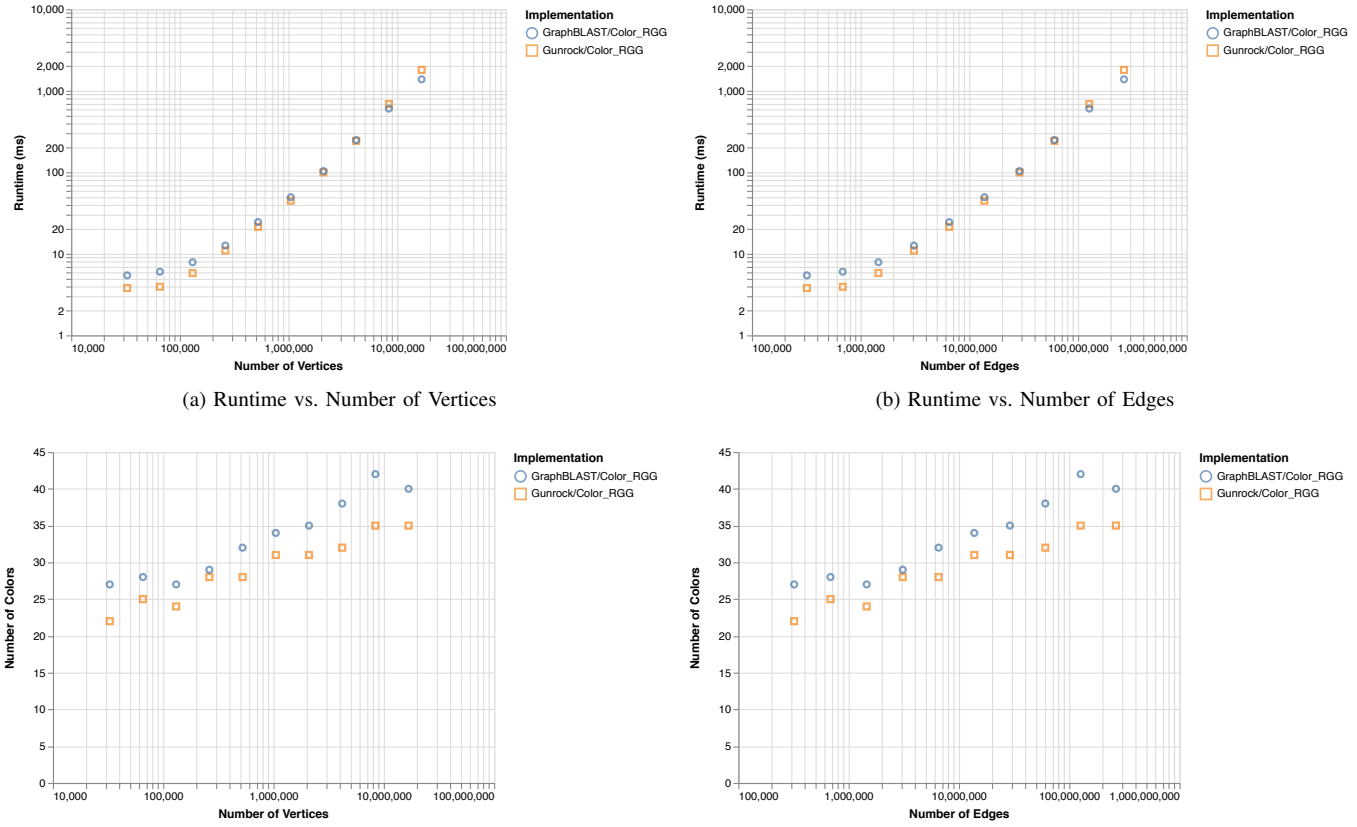
[9] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your Jacobian? Graph coloring for computing derivatives," *SIAM Review*, vol. 47, no. 4, pp. 629–705, 2005.

[10] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, 2017.

[11] C. Yang, "GraphBLAST library," http://github.com/gunrock/graphblast, 2015.

[12] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-LU factorization on the GPU," NVIDIA Research, Tech. Rep. NVR-2015-001, May 2015.

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



(a) Runtime vs. Number of Vertices

(b) Runtime vs. Number of Edges

(c) Number of Colors vs. Number of Vertices

(d) Number of Colors vs. Number of Edges

Fig. 3: Runtime and number of colors computed by our Gunrock and GraphBLAST implementations as a function of sizes (vertex and edge counts) in RGG-generated graphs.

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

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

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

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

[18] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," *Northeast Parallel Architecture Center, Syracuse University, Tech. Rep*, 1995.

[19] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek, "A framework for scalable greedy coloring on distributed-memory parallel computers," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[20] S. Salihoglu and J. Widom, "Optimizing graph algorithms on Pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 577–588, 2014.

[21] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on GPUs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 297–298, 2011.

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

[23] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[24] W. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph programming interface (GPI): A linear algebra programming model for large scale graph computations," *International Journal of Parallel Programming*, vol. 46, no. 2, pp. 412–440, 2018.

[25] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[26] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, *The GraphBLAS C API Specification*, Nov. 2017, rev. 1.1.

[27] ——, "Design of the GraphBLAS API for C," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2017, pp. 643–652.

[28] C. Yang, A. Buluç, and J. D. Owens, "Implementing push-pull efficiently in GraphBLAS," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP 2018, Aug. 2018, pp. 89:1–89:11.

[29] T. Mattson, C. Yang, S. McMillan, A. Buluç, and J. Moreira, "GraphBLAS C API: Ideas for future versions of the specification," in *IEEE High Performance Extreme Computing Conference*, 2017, pp. 1–6.

[30] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering*. American Mathematical Society, 2012.

[31] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *IEEE International Symposium on Parallel & Distributed Processing*, 2010, pp. 1–12.

[32] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1:1–1:25, Nov. 2011.