Graph Coloring using GPUs

Double Blind

No Institute Given

Abstract. Graph coloring is a widely studied problem that is used in a variety of applications, such as task scheduling, register allocation, eigenvalue computations, social network analysis, and so on. Many of the modern day applications deal with large graphs (with millions of vertices and edges) and researchers have exploited the parallelism provided by multi-core systems to efficiently color such large graphs. GPUs provide a promising parallel infrastructure to run large applications. In this paper, we present new schemes to efficiently color large graphs on GPUs.
We extend the algorithm of Rokos et al. [21] to efficiently color graphs using GPUs. Their approach has to continually resolve conflicts for color assignment. We present a data driven variation of their algorithm and use an improved scheme for conflict resolution. We also propose two optimizations for our algorithm to reduce both the execution time and memory requirements. We have evaluated our scheme (called SIRG) against the NVIDIA cuSPARSE library and the work of Chen et al. [13], and show that SIRG runs significantly faster: $3.42\times$ and $1.76\times$, respectively.

1 Introduction

Graph Coloring, widely studied as vertex coloring in an undirected graph, refers to the assignment of colors to the vertices of a graph such that no two adjacent vertices are assigned the same color. It is used in various applications such as scheduling of tasks [16], register allocation [4], eigenvalue computations [15], social network analysis [6], sparse matrix computations[12], and so on. The problem of optimal graph coloring and even that of finding the *chromatic number* of a graph (minimum number of colors needed to color the graph) are NP-Hard. Hence, various heuristics have been proposed to solve the graph coloring problem. As many modern applications deal with graphs containing millions of vertices and edges, coloring of such large graphs sequentially leads to prohibitively high execution times. To address this issue, various parallel graph coloring algorithms have been designed for multi-core and many-core systems.

Though graph coloring as a problem has been solved using many heuristics [2,8,20,17], parallel graph coloring algorithms have mostly been extensions of two main approaches: (1) Maximal Independent Set (MIS) approach that finds maximal independent sets of a graph and assigns a unique color to each independent set. (2) Greedy approach that assigns each vertex v the smallest color that has not been assigned to any adjacent vertices of v.

Luby [14] proposed one of the first parallel graph coloring algorithms by computing MIS in parallel. The algorithm was later extended by Jones and

Plassmann [11]. Compared to the MIS based algorithms, owing to the simplicity in implementation and ease of parallelization of greedy algorithms, many researchers have proposed the greedy approach based parallel graph coloring algorithms. For example, Gebremedhin and Manne [9] proposed a three-step approach as the initial parallelization of the sequential greedy algorithm. In Step1, the algorithm colors all the vertices in parallel with the minimum available color (that smallest color that has not been used to color any of the adjacent vertices). This may lead to conflicts between adjacent vertices. In Step2, conflicts are detected and for each conflict one of the vertices retains the color and the other vertex loses its color. After resolving conflicts, in Step3, all the remaining uncolored vertices are colored with the minimum available colors, sequentially. The Step3 was parallelized by Çatalyürek et al. [3], by invoking Step1 and Step2 on the remaining uncolored vertices, repeatedly. The process continues until there are no conflicts. After each invocation of Step1 and Step2, a barrier is inserted to synchronize among the threads.

Rokos et al. [21] improved the algorithm of Çatalyürek et al. [3] by reducing the synchronization overheads among the threads. In case of conflicts, instead of uncoloring and recoloring, the algorithm recolors the conflicting-vertex in the same iteration with the minimum available color. We refer to this improvised algorithm as RIG (Rokos Improvised Greedy).

As GPUs provide massive amounts of parallelism and are widely being used to run algorithms on large datasets, there also have been efforts to design parallel algorithms that can run efficiently on GPUs. For example, the csrcolor function, in the cuSPARSE library [19] of NVIDIA, implements the parallel MIS algorithm for GPUs. Grosset et al. [10] presented the first implementation of the greedy algorithm of Gebremedhin and Manne [9] on GPUs. Recently, Chen et al. [13] extended the work of Çatalyürek et al. [3] on to GPUs with a few optimizations; we refer to their work as ChenGC. One main drawback of their work is that their algorithm needs a pre-set value of the maximum color required ($maxColor$) to color the graph and the algorithm does not terminate if the value of $maxColor$ is too low. In contrast, setting $maxColor$ to a very high value leads to very high execution times and memory usage. Though the NVIDIA's cuSPARSE library does not suffer from any such limitations, it uses a large number of colors for producing a valid coloring of the graph. And also it runs slower (55% [13]) than ChenGC. In this paper, we present a solution to address these limitations.

We extend the algorithm of Rokos et al. [21] to efficiently color graphs using GPUs. We provide a data-driven extension of their algorithm, along with new heuristics for faster executions. We propose two optimizations for improving both the execution time and memory requirements. We have evaluated our optimized algorithm (referred to as SIRG – $\underline{S}$calable and $\underline{I}$mproved $\underline{R}$IG Algorithm for $\underline{G}$PUs) and found that SIRG runs 3.42× and 1.76× faster than csrcolor and ChenGC, respectively. We have also studied the impact of our proposed optimizations and the various design decisions and found them to be effective.

```
1  Function RIG ($G$) // $G = (V, E)$
2  begin
3      $U = V$;
4      foreach $v \in U$ do // parallel loop              }
5          $C = \{$colors of $u \in adj(v)\}$;            } Coloring Phase
6          color($v$) = minimum color $c \notin C$;       }
7      barrier();
8      while $|U| > 0$ do                                                          }
9          $L = \phi$;                                                             }
10         foreach $v \in U$ do // parallel loop                                   }
11             if $\exists\ v' \in adj(v),\ v' > v$: $color(v) == color(v')$ then  } Conflict Re-
12                 $C = \{$colors of $u \in adj(v)\}$;                             } solve and
13                 color($v$) = minimum color $c \notin C$;                        } Recolor Phase
14                 $L = L \cup \{v\}$;                                             }
15         barrier();                                                              }
16         $U = L$;                                                                }
```

Fig. 1: Improvised Greedy Algorithm of Rokos et al. [21].

2 Background

Algorithm of Rokos et al. For the sake of completeness, we briefly present the improvised greedy algorithm of Rokos et al. [21] (Fig. 1). We refer to it as the RIG (Rokos Improvised Greedy) algorithm. It consists of two phases: the `Coloring` phase (lines 4-6) and the `ConflictResolveAndRecolor` phase (lines 8-16) with a barrier in between, for synchronization. The `Coloring` phase tentatively assigns (in parallel) every vertex a color based on the *minimum available color* (the smallest color that is not assigned to any of its neighbouring vertices). After every vertex has been processed, the `ConflictResolveAndRecolor` phase starts, where every vertex v is checked for conflict of colors with its neighbouring vertices that have vertex-number higher than that of v. If a conflict is detected, the vertex with higher vertex number retains the color. The vertex with a lower vertex number is recolored by checking for colors of its neighbours and assigning the minimum available color. Once all the vertices have been processed (enforced by a barrier), the phase continues with the recolored vertices. The algorithm terminates when no vertices have to be recolored. This, in turn, indicates that the graph vertices have a valid coloring.

Parallelization on GPU. In CUDA programs, the computation is typically divided between the host (CPU) and device (GPU). The host side computation includes the allocation of the required memory on the device and copying of data required by the program from the host to the device. The host also launches the device code using a command like $\lll M, N \ggg$`kernelFunc()`, to launch M number of threads on each of the N thread-blocks; the values of M and N are set by the programmer. After the parallel execution of the kernels, the control returns to the host. The required data is copied back to the host from the device.

3 Graph Coloring for GPUs

In this section, we present our novel graph coloring algorithm that can be run efficiently on GPUs. We derive this algorithm from the insightful work of Rokos et al. [21] (described in Section 2). We first show why their argument about the non-termination of their algorithm (for GPUs) does not hold. Then we extend their algorithm with a few heuristics for efficient execution on GPUs.

3.1 Non-termination of the RIG Algorithm

Rokos et al. [21, Section 5] discuss that the algorithm in Fig. 1 goes into an infinite loop due to SIMT-style execution of GPU threads. However, the algorithm will not lead to an infinite loop if the comparison at Line 11 is based on some unique ids (such as vertex numbers), which ensures that no two adjacent vertices will keep flipping their colors forever (as alluded by Rokos et al.). In this paper, we maintain and use unique vertex ids for such conflict resolution.

3.2 Improvements to RIG

We now list two improvements to the RIG algorithm (Section 2). The first one improves the conflict resolution criteria, and the second one is an efficient mechanism to implement the algorithm for GPUs.

Conflict Resolution. For the ease of presentation, for each vertex v, we use $S(v)$ to denote the set of neighbouring vertices that need to be checked for conflicts in every iteration (Line 11, Fig. 1). Fig. 1 resolves the conflicts by giving priority to the higher number vertex (Line 11) and uses $S(v) = \{u | u \in adj(v), u > v\}$. While this works as a fine criterion for avoiding infinite loops, it can be improved by using the degree of the nodes as the first criteria for conflict resolution. We set $S(v) = \{u | u \in adj(v), degree(u) > degree(v)) || (degree(u) == degree(v) \,\&\&\, u > v)\}$. Thus, $S(v)$ includes the set of adjacent vertices of v, such that either their degree is greater than that of v, or they have the same degree as v, but have higher vertex-number than v. The intuition of setting $S(v)$ by using a prioritization scheme based on the degree is that it will lead to fewer conflicts, as the vertices with higher degrees will be removed from contention early. Note that we still include the vertex number based check to ensure that the algorithm does not go into an infinite loop.

Data-driven implementation. We use the data-driven method proposed by Nasre et al.[18] to realize an efficient implementation of the RIG algorithm (Fig. 1) for GPUs. The original algorithm has two parts: (i) the coloring phase (lines 4-6) and (ii) the conflict-resolve-and-recolor phase (lines 8-16). Our data-driven implementation mainly improves the second part.

```
Function GraphColoring ($G$) // $G = (V, E)$
begin
    $\lll M, N \ggg$Coloring($G$);
    barrier();
    $W_{in} = V$;
    while $W_{in} \neq \phi$ do
        $\lll M, N \ggg$ConflictResolveAndRecolorKernel($G, W_{in}$);   } Conflict Re-
        barrier();                                                          } solve and
        swap($W_{in}$, $W_{out}$);                                          } Recolor Phase
```

Fig. 2: Data Driven Implementation - CPU

```
Function Coloring ($G$) // $G = (V, E)$
begin
    for $vertex\ v \in V|myThread$ do
        $C$ = {colors of $u \in adj(v)$};
        color($v$) = minimum color $c \notin C$
    return color
```

Fig. 3: Data driven implementation. Coloring Phase on GPU

Fig. 2 shows the main pseudocode to be executed on the host (CPU). The initial `Coloring` phase (see Fig. 3) is similar to the RIG algorithm, except that $M \times N$ GPU threads are launched. Each GPU thread is assigned a set of vertices to be colored (shown by the projection $V|myThread$).

The next phase (conflict-resolve-and-recolor) maintains two shared worklists W_{in} and W_{out}, where W_{in} represents the vertices that still need to be recolored. Initially, W_{in} contains the list of all the vertices. In every iteration, the host launches the GPU kernel on a set of GPU threads. Each of the GPU threads runs the code shown in the function `ConflictResolveAndRecolorKernel` (Fig. 4).

Each thread picks a vertex from the list of vertices (from W_{in}) that are assigned to it (represented by the projection $W_{in}|myThread$). In our implementation, we have used a blocked-cyclic distribution. Each vertex is checked for conflicts based on the conflict-resolution heuristic discussed above. In case a conflict is detected, the vertex is recolored with the minimum available color, and the vertex is added to W_{out}. Since W_{out} is a shared list across the GPU threads, this operation has to be done atomically. See Section 5 on how we implement it efficiently. If no conflict is detected for a vertex, then the vertex retains its color and is not considered for (re)coloring in the subsequent iterations.

On the host, at the end of every iteration of the while-loop, W_{in} and W_{out} are swapped (double buffering [18]) and the process continues. The algorithm terminates when W_{in} does not contain any more vertices; that is, all the vertices have been colored without any conflicts. Thus, the graph finally has a valid coloring at the end of the algorithm.

```
1 Function ConflictResolveAndRecolorKernel ($G$, $W_{in}$)
2 begin
3     $W_{out} = \phi$;
4     for $v \in W_{in}|myThread$ do
5         if $\exists\ v' \in adj(v),\ v' \in S(v)$: $color(v) == color(v')$ then
6             $C = \{\text{colors of } u \in adj(v)\}$;
7             $\text{color}(v) = \text{minimum color } c \notin C$ ;
8             $W_{out} = W_{out} \cup \{v\}$;                // Atomic operation
```

Fig. 4: Conflict resolution and recoloring phase on GPU.

4 Optimizations

We now list two optimizations for the baseline algorithm discussed in Section 3. Both these optimizations are related to the efficient implementation of the data structure that holds the set of colors of the adjacent vertices. We denote the baseline algorithm of Section 3 along with the optimizations discussed in this section, as SIRG (Scalable and Improved RIG Algorithm for GPUs).

In the GPU algorithm shown in Figures 3 and 4, every thread colors/recolors a vertex with the minimum available color. For this, a naive way of implementation would be to use, for each vertex, an integer array `adjColors` to hold one bit for each of the colors that might be required to color the graph. Hence, the size of `adjColors` = $\lceil (maxColor \div 32) \rceil$, where $maxColor$ is the estimated maximum number of colors required to color the graph. As a quick and conservative estimate, we use the following equation as the estimate for $maxColor$.

$$maxColor = 2^{\lceil (\log_2 (1+\text{maximum-degree-of-the-graph})) \rceil} \tag{1}$$

For every vertex v, initially, every bit of the array `adjColors` is set to 1. For every adjacent vertex of v, the bit corresponding to the color of that adjacent vertex is unset in `adjColors`. Then, the color corresponding to the first bit in `adjColors` that is set to 1, is assigned to v.

Considering the overheads of maintaining, for each vertex, an individual `adjColors` array, and the scalability issues thereof, we allocate one `adjColors` array for each thread on the GPU device. An important point to note is that every thread may loop over all the elements of the `adjColors` array twice, for every vertex in every iteration – finding the first set bit (to find the min color, line 7, Fig. 4) and for resetting the array (to all 1s, at the end of processing each vertex, after line 8, Fig. 4). Hence, the size of the `adjColors` has a significant impact on the execution time of the individual threads and consequently the overall kernel. We now discuss two optimizations to address this challenge.

4.1 Use of `long long int` and CUDA `__ffsll` Instruction

CUDA provides a hardware instruction `__ffsll` to find the first set bit in a `long long int` number. Hence, we can use a `long long int` `adjColors` array (instead of an `int` array), where each element of the array can hold 64 bits – corresponding to 64 colors. The size of the array would be reduced to $maxColor \div 64$, from $maxColor \div 32$. As the size of the array decreases by half, every thread needs to loop a fewer number of times over the `adjColors` array, for every vertex thereby improving the performance.

Considering that this optimization is useful only when the initial number of colors is > 32, we use two versions of the code (one using `__ffsll` and one without); one of them is invoked at runtime, depending on the maximum degree of the input graph.

4.2 Stepwise Doubling of maximum colors required

As discussed before, we use $maxColor$ to compute the estimate for the size of `adjColors`. However many of the web graphs are usually sparse graphs, with a small number of vertices having large degrees and the rest having low degrees. Consequently, using equation (1), we end up setting $maxColor$ to a unnecessarily high value, even though the actual number of colors required to color the graph is relatively very small. Such high values for $maxColor$ increase the size of the `adjColors` array, thereby increasing the execution time (and increasing the memory requirements). We now present a scheme to reduce these overheads.

We set the initial value of $maxColor$ to be a small number K_0. Consequently, the initial size of the `adjColors` array will be small, but may not be enough to color the graph without any conflicts. This insufficiency in the number of colors can be detected when there is no bit set (color available) in the `adjColors` array in an iteration (line 5 in Fig. 3 and line 7 in Fig. 4). In such a case, we double the value of $maxColor$ and resize the `adjColors` array, and continue the coloring process for the remaining vertices. Such a resizing can happen till the $maxColor$ value is sufficient to color the graph without any conflicts.

By doubling the value of $maxColor$ when required and not setting it to a large value conservatively, the size of the `adjColors` array can be significantly reduced. In our evaluation, we use $K_0 = min\left(256, 2^{\lceil(\log_2(1+\text{maximum-degree-of-the-graph}))\rceil}\right)$. Thus, this optimization is impactful only for graphs whose maximum degree is > 256.

5 Discussion

We now present some of the salient points in the implementation of SIRG.

Compressed Sparse Row. We represent the graph using the standard compressed sparse row (csr) format, that uses two arrays (ColIndices and Offset) to represent the graphs. In addition, we maintain another array (called nextVertices) to efficiently find, for each vertex v, the set of neighboring vertices that need to be checked for conflicts, (given by $S(v)$, see Section 3). The element nextVertices[i] points to the index in ColIndices such that the vertices in adjacency list of v_i from nextVertices[i] to Offset[i+1] belong to $S(v_i)$. The ColIndices array is arranged such that, for each vertex v_i, the vertices in the adjacency list of v_i are ordered by the vertex number, or the vertex degree, depending on whether vertex number or degree is used for conflict

Network	Nodes (10^6)	Edges (10^6)	Avg degree	Type
EUROPE_OSM	50.9	108.1	2.12	Road Network
ROAD_USA	23.9	57.7	2.41	Road Network
ORKUT	3.1	234.3	76.28	Scale Free Network
LIVEJOURNAL	3.9	69.4	17.35	Scale Free Network
TWITTER7	41.6	323.3	7.76	Scale Free Network
MYCIELSKIAN19	0.4	903.2	2296.95	General Network
RMAT_1	10.0	199.9	20.00	Synthetic (0.5,0.5,0.5,0.5)
RMAT_2	20.0	809.9	40.49	Synthetic (0.1,0.3,0.4,0.2)

Fig. 5: Graphs used in our experiments

resolution (see Section 3). Maintaining this additional array nextVertices can provide access to the elements of $S(v_i)$ in $\mathcal{O}(1)$ time during the conflict resolution phase.

Adding elements to worklist. In Fig. 4 (Line 8), every thread updates a shared worklist (W_{out}) and this has to be done atomically. This leads to the invocation of a large number of atomic operations, which can be potentially inefficient. To address this issue we use the popular idea of using prefix sum to find the appropriate indices where each thread of a warp can write (in parallel) to the shared worklist, independently of each other. This leads to execution of one atomic operation per warp (in contrast to one atomic operation per thread) and hence reduces the number of atomic operations by a factor of up to 32 (warp size).

Distribution of worklist elements among GPU threads. The elements of the worklist W_{in} have to be divided among GPU threads in a data-driven implementation. For efficiency, we implemented the worklists (W_{in} and W_{out}) as global arrays and these elements are distributed among the GPU threads in a cyclic order. Therefore, a thread with id $= t$, accesses the vertices from W_{in} such that the index i of the vertex in W_{in} satisfies the equation $t = i\%totalNumThreads$. Note that we have also tried using the blocked distribution, but found the cyclic distribution to be more efficient.

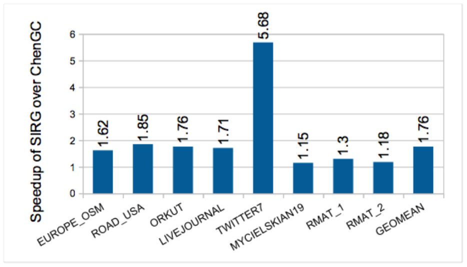
Number of threads. The optimal number of blocks per SM (*maximum residency*) to be launched depend on many factors, such as the blocksize, number of registers, shared memory size, and so on. We set the total number of blocks launched to be equal to maximum residency $\times$ number of SMs. On experimentation, we found that setting blocksize $=$ 1024 threads gave the best performance, on our NVIDIA P100 system.

Topology-Driven Implementation. In addition to the data-driven implementation discussed in Section 3, we also implemented the coloring algorithm of Rokos et al. [21] using the topology-driven method [18]. We observed that the topology-driven implementation was significantly slower than the data-driven implementation and hence not elaborated on, in this manuscript.

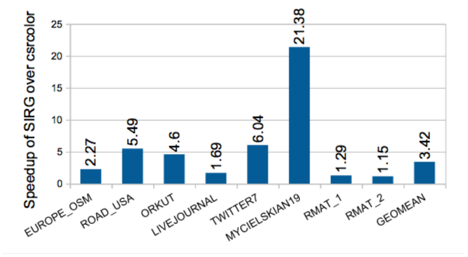
Difference in memory requirements between SIRG and ChenGC. Both SIRG and ChenGC [13] follow the greedy approach and the memory usage is similar. Compared to ChenGC, SIRG uses only 16 bytes extra to maintain some additional metadata information.

6 Implementation and Evaluation

We have compiled our codes using the CUDA 9.1 compiler and executed them on a Tesla P100 GPU, with 12GB memory. We have evaluated our codes using eight differ-



(a) Speedup over ChenGC.



(b) Speedup over csrcolor.

Fig. 6: Speedup of SIRG over ChenGC and csrcolor.

ent graph inputs (details shown in Fig. 5), with vertices varying between 0.4M to 50M, edges varying between 57M to 903M. These inputs span both real-world graphs like road networks and scale-free networks that follow power-law, and synthetic graphs (last three). While the first six are obtained from the Florida Sparse Matrix Collection [7], the last two are created using the R-MAT [5] graph generator (using the parameters shown in Fig. 5). We now present our evaluation to understand (i) the performance improvements realized by SIRG (uses the schemes discussed in Sections 3 and 4) over the existing graph coloring algorithm of Chen et al. [13] that is targeted to GPUs (abbreviated ChenGC), and NVIDIA's cuSPARSE library. (ii) the effect of the proposed optimizations, and (iii) the impact of some of the design decisions. All these codes (different versions of SIRG, ChenGC, and the code using the cuSPARSE library) used for the comparative evaluation can be found on GitHub [1].

6.1 Comparison of SIRG Vs ChenGC and csrcolor

To perform a comparative evaluation of SIRG, we used ChenGC and the NVIDIA's cuSPARSE library (csrcolor function) to color the input graphs. While csrcolor was general enough to color any given graph, we found that ChenGC did not terminate for four of the eight input graphs. We found that the issue was because ChenGC uses a fixed value for the maximum number of the required colors ($maxColor$) – the value for this variable is hardcoded in their algorithm, unlike in SIRG, where no such restriction is present. We found that in ChenGC, while setting $maxColor$ to a very large number, made the programs run successfully on all the inputs, but it had a drawback – the programs took a very long time to run. On experimentation, we found the minimum value for the variable $maxColor$, in order for ChenGC to run successfully on all the input graphs was 1024; hence, we set $maxColor$=1024 in ChenGC.

Fig. 6a shows the speedup of SIRG with respect to ChenGC in terms of execution time. We can observe that across all the input graphs, SIRG performs better than ChenGC (between 1.15× to 5.68×, geomean 1.76×).

We find that in the real-world graphs the gains are much more than that in the synthetic graphs. In general, we found that the stepwise-doubling optimization was most effective in these real-world graphs in improving the performance. And this impact was much higher in power-law graphs (for example, TWITTER7).

Fig. 6b shows the speedup of SIRG over csrcolor. In contrast to ChenGC, we did not have to make any changes to csrcolor, for it to run. Fig. 6b shows that SIRG performs

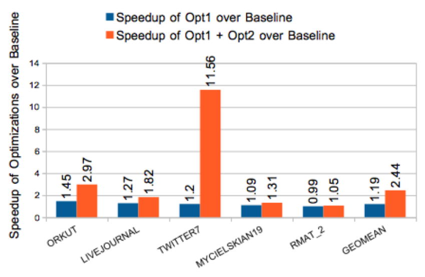


Fig. 7: Effect of the optimizations.

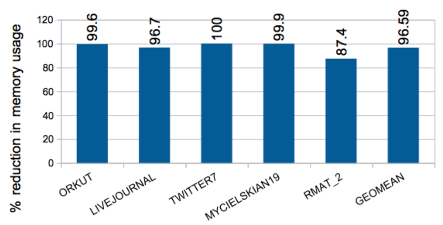


Fig. 8: Effect on memory usage due to stepwise-doubling optimization.

significantly better than csrcolor (1.15× to 21.38×, geomean 3.42×). We see that except for RMAT_1 and RMAT_2, SIRG performs leads to remarkably higher performance across both power-law and non-power-law graphs. We believe the very high speedups obtained in MYCIELSKIAN19 is because of the specific nature of the input graph (higher density, total number of nodes = 0.4 million, average degree ≈ 2300), which is making the csrcolor perform poorly.

We have also compared the coloring quality (number of colors used) by the three algorithms under consideration. While SIRG uses significantly fewer number of colors (geomean 77% less) than csrcolor, the number is comparable to that of ChenGC (geomean difference $< 6\%$). We have done some experiments (not shown here) and observed that even this minor difference is mainly related to the order in which the threads process the vertices.

Summary. We see that SIRG performs significantly better than csrcolor. It even performs better than ChenGC, which has to be tuned manually in order to run successfully on various graph inputs.

6.2 Impact of the proposed optimizations

Fig. 7 shows the effect of the two proposed optimizations (Opt1: Section 4.1, and Opt2: Section 4.2) over our baseline approach (Section 3); the graph shows the achieved speedup over input graphs where the optimizations were invoked. For EUROPE_OSM, ROAD_USA and RMAT_1 where the maximum degree was not more than 32, Opt1 was not invoked, and Opt2 had no effect.

We see that, for most inputs, Opt1 performs better than the Baseline and Opt2 adds to the performance improvements much more. We also observe that in the power-law graphs, the effect of Opt2 is high and led to large gains (up to 11.56×).

In Section 4.2, we discuss that Opt2 (stepwise-doubling optimization) can also help reduce memory consumption. We show this impact in Fig. 8. The figure compares the memory consumption of SIRG, against SIRG without Opt2, for the inputs on which Opt2 had some impact. It shows that the impact of Opt2 on the memory requirements is high: leads to geomean 96.59% reduction in memory.

Summary. Our evaluation shows that the proposed optimizations lead to significant gains and attests to the importance of these optimizations.

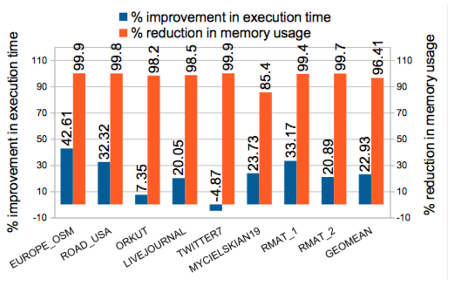


Fig. 9: %Improvement due to per-thread Vs per-vertex allocation of `adjColors`.

6.3 Impact of maintaining `adjColors` array per thread

In Section 4, we have discussed that due to the memory overheads and scalability issues, we allocate `adjColors` array for each thread instead of each vertex. We now discuss the impact of such a choice. We show the impact (in Fig. 9) in terms of total time and memory usage for two configurations: (i) SIRG with `adjColors` array allocated for each vertex, and (ii) default SIRG: with `adjColors` array allocated for each thread.

The figure shows that allocating `adjColors` for each vertex can increase the memory requirements significantly, which is avoided by doing per thread allocation (Geomean 96.41%). While a per-vertex scheme may lead to some minor gains for some inputs (for example, 4.87% for TWITTER7), overall we find that per-thread allocation of `adjColors` led to better execution times (geomean 22.93%).

We found that allocating `adjColors` per vertex increases the memory requirement so much that in the absence of Opt2 (which reduces the memory consumption significantly), the program runs out of memory for many inputs (for example, ORKUT, TWITTER7, MYCIELSKIAN19). This further shows the importance of our choice of per-thread allocation of the `adjColors` array.

Overall summary. Our evaluation shows that SIRG performs better than both csrcolor and ChenGC. We found our optimizations and design choices lead to efficient executions (both in terms of execution time and memory usage).

7 Conclusion and Future work

In this paper, we presented a fast and scalable graph coloring algorithm for GPUs. We extended the algorithm by Rokos et al. [21] to efficiently color graphs for GPUs using a data parallel implementation, with a better heuristics for color-conflict resolution. We also proposed two optimization techniques to improve both the execution time and memory requirements. We showed that compared to the NVIDIA's cuSPARSE library and the work of Chen et al. [13], our implementation runs 3.42× and 1.76× faster, respectively.

References

1. Anonymous. Reference Implementations of the Register Allocation Algorithms. Supplementary Material, 2019. https://github.com/incognito-anonymous/GraphColoringUsingGPUs.

2. Norman Biggs. Some heuristics for graph colouring. In Roy Nelson and Robin J. Wilson, editors, *Graph Colourings*, pages 87–96. 1990.
3. Ü. V Çatalyürek, J. Feo, A. H Gebremedhin, M. Halappanavar, and A. Pothen. Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures. *Parallel Computing*, 38(10-11):576–594, 2012.
4. G. J Chaitin. Register Allocation & Spilling via Graph Coloring. In *ACM SIGPLAN Notices*, volume 17, pages 98–105. ACM, 1982.
5. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *ICDM*, pages 442–446. SIAM, 2004.
6. D. Chalupa. On the Ability of Graph Coloring Heuristics to Find Substructures in Social Networks. *Information Sciences and Technologies, Bulletin of ACM Slovakia*, 3(2):51–54, 2011.
7. T. A Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM TOMS*, 38(1):1, 2011.
8. J.-K. Dorne, R.and Hao. A New Genetic Local Search Algorithm for Graph Coloring. In *International Conference on Parallel Problem Solving from Nature*, pages 745–754. Springer, 1998.
9. A. H. Gebremedhin and F. Manne. Scalable Parallel Graph Coloring Algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.
10. A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall. Evaluating Graph Coloring on GPUs. *ACM SIGPLAN Notices*, 46(8):297–298, 2011.
11. M. T Jones and P. E Plassmann. A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.
12. M. T Jones and P. E Plassmann. Scalable Iterative Solution of Sparse Linear Systems. *Parallel Computing*, 20(5):753–773, 1994.
13. P. Li, X. Chen, Z. Quan, J. Fang, H. Su, T. Tang, and C. Yang. High Performance Parallel Graph Coloring on GPGPUs. In *IPDPS Workshops*, pages 845–854. IEEE, 2016.
14. M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *Journal on Computing*, 15(4):1036–1053, 1986.
15. F. Manne. A Parallel Algorithm for Computing the Extremal Eigenvalues of Very Large Sparse Matrices. In *International Workshop on Applied Parallel Computing*, pages 332–336. Springer, 1998.
16. Dá. Marx. Graph Colouring Problems and their Applications in Scheduling. *Periodica Polytechnica Electrical Engineering*, 48(1-2):11–16, 2004.
17. A. Mehrotra and M. A. Trick. A Column Generation Approach for Graph Coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996.
18. R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. In *IPDPS*, pages 463–474. IEEE, 2013.
19. CUDA Nvidia. CuSPARSE Library. *NVIDIA Corporation, Santa Clara, CA*, 2014.
20. WJM Philipsen and L. Stok. Graph Coloring using Neural Networks. In *IEEE International Sympoisum on Circuits and Systems*, pages 1597–1600. IEEE, 1991.
21. G. Rokos, G. Gorman, and P. HJ Kelly. A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures. In *EuroPar*, pages 414–425. Springer, 2015.