CSC 458 Final Project Report
Parallel Graph Coloring Algorithm with CUDA

Minh Dang

April 2024

1 Introduction

This project attempts to solve the Graph Coloring Problem [3] using parallelization. The problem is stated as follows:

> Given a graph $G(V, E)$, find a coloring $c : V \to N$ such that no two adjacent vertices have the same color.

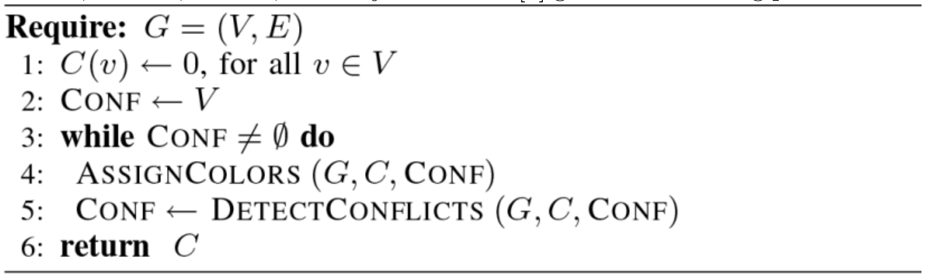
This problem assumes the graph to be non-directed. Of course, assigning $|V|$ colors to $|V|$ vertices would be the most obvious solution. The ultimate goal here is to find the coloring with the least number of colors used.

Program input: The first line of the input is the number of vertices (n) and edges (m). Each of the next m lines consists of two numbers indicating the two vertices that are ends of an edge.

Program output: An assignment of a color to each vertex of the graph.

2 Algorithms

Deveci, Boman, Devine, and Rajamanickam [1] gives the following pseudo-code:



```
Require: $G = (V, E)$
1: $C(v) \leftarrow 0$, for all $v \in V$
2: CONF $\leftarrow V$
3: while CONF $\neq \emptyset$ do
4:   ASSIGNCOLORS (G, C, CONF)
5:   CONF $\leftarrow$ DETECTCONFLICTS (G, C, CONF)
6: return C
```

This pseudo-code represents the Iterative Graph Coloring Algorithm [1]. Colors are assigned to vertices until no conflicts are detected.

The step AssignColors can be parallelized as such [1]:

```
Require: $G = (V, E), C,$ Conf
1: Allocate private Forbidden with size max degree.
2: for $v \in$ Conf in parallel do
3:   Forbidden $\leftarrow false$
4:   Forbidden$(C(u)) \leftarrow true$ for $u \in adj(v)$
5:   C(v) $\leftarrow \min \{i > 0 |$Forbidden$(i) = false\}$
6: return $C$
```

Forbidden is a boolean array to record which colors a vertex cannot be assigned to. A vertex is colored to the minimum color that is not forbidden, ensuring that the total number of colors used is the lowest possible.

The step DetectConflicts can also be parallelized as such [1]:

```
Require: $G = (V, E), C,$ Conf
1: NewConf $\leftarrow \emptyset$
2: for $v \in$ Conf in parallel do
3:   for $u \in adj(v)$ do
4:     if $C(u) = C(v)$ and $u < v$ then
5:       Atomic NewConf $\leftarrow$ NewConf $\cup v$
6: return NewConf
```

NewConf is an array of vertices that are conflicting with each other and is updated with an atomic function. The additional condition of $u < v$ means only the larger conflicting vertex is added to NewConf. This means conflicts are resolved on the basis of vertex value: low vertex's color is fixed, high vertex's color is changed.

3 Parallelization & Implementation Details

This project uses NVIDIA CUDA [4] to manage parallelization. The general idea is to implement the iterative algorithm, then wrap the parallelizable parts with CUDA code. More specifically, the code for AssignColors and DetectConflicts in CUDA device code is quite similar to the iterative algorithm, then the results are synchronized by CUDA host code in wrapper functions.

The code package comes with a random graph generator in C++. An user can choose how many vertices and edges they want on their randomly generated graph. This program produces an output file which serves as the input to the CUDA code file.

The code package also comes with a visualization method using Java Graphics. This program assigns the color number in the CUDA output file to visual colors and draws a graph with vertices at random coordinates.

Project Running Steps:

1. Run the random graph generator file to produce a randomly generated graph file, named `graph.in`.
2. Run the CUDA code file with `graph.in` as the input, receive the coloring recorded in output file named `coloring.in`.
3. Run the Java Graphics file with `graph.in` and `coloring.in` and produce the visualization of the colored graph.

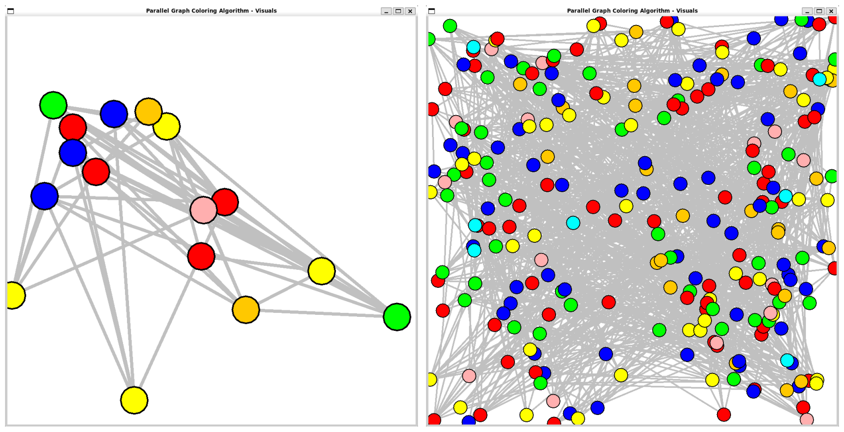
4 Experiment Design & Results

4.1 Experiment Design

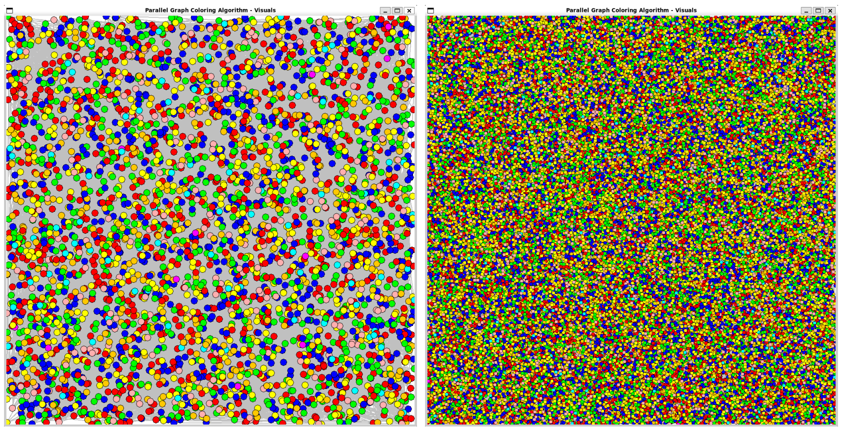
The experiment on the code package is designed to optimize the parallelization with CUDA, testing from 2^5 to 2^{10} threads per block. The experiment also tests with multiple graph sizes from 2^4 to 2^{16} vertices (with the number of edges being 2^2 times the number of vertices).

4.2 Experiment Results

Example results visualization:

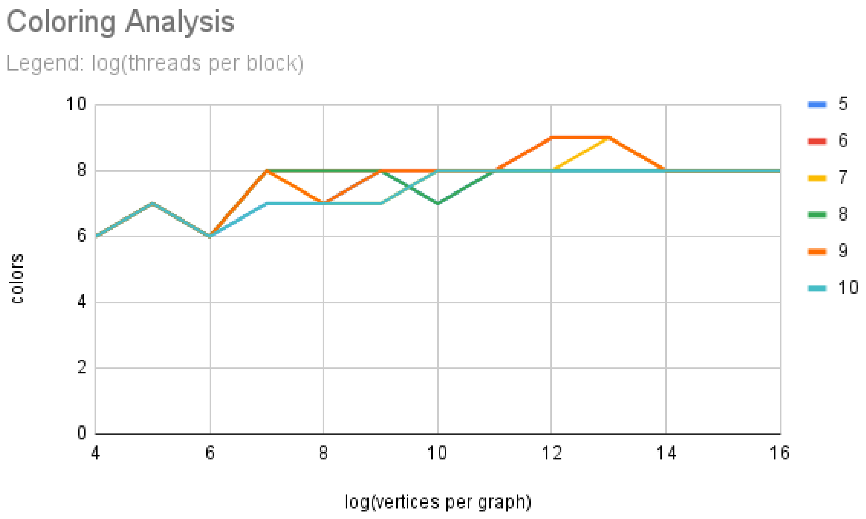

Parallel Graph Coloring Algorithm - Visuals
Parallel Graph Coloring Algorithm - Visuals

$n = 2^4, c = 6$ and $n = 2^8, c = 7$


Parallel Graph Coloring Algorithm - Visuals
Parallel Graph Coloring Algorithm - Visuals
$n = 2^{12}, c = 9$ and $n = 2^{16}, c = 8$

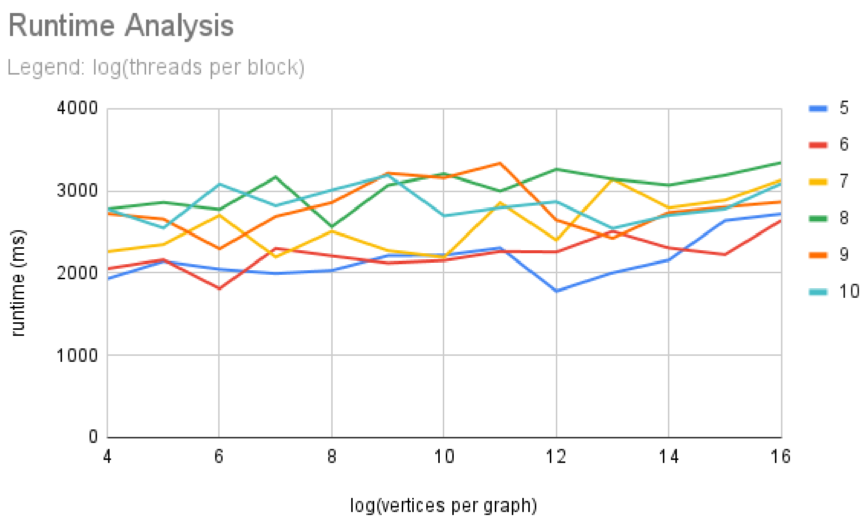
4.3 Results Analysis

The coloring analysis are presented in the following graph:



Despite the number of threads per block, the number of colors used is consistent with the growth of the number of vertices per graph. This observation of consistency also proves the correctness of the algorithm. The number of colors used ranges from 6 to 9, which is significantly less than the number of vertices per graph.

The runtime results (in milliseconds) are presented as follows:



The runtime seems to fluctuate randomly when graph size increases. This happens for every number of threads per block in the testing range.

5 Discussion

The coloring analysis shows that this algorithm works well. Nevertheless, the existence of the Four Color Theorem [2] implies that this algorithm can be refined, or that there are better algorithms out there, so that the minimum number of colors used can be reduced to 4.

The runtime analysis, however, shows results dissimilar to the author's prediction. The author predicted that at larger graph sizes, more threads per block means lower runtime (which is equivalent to higher speed-up rates). One possible explanation for the fluctuating runtime is that at graph sizes this small, the processing speed and the expense of multi-threading are cancelling out. At larger graph sizes, multi-threading might prove its power and gives higher speed-up rates.

There are a plethora of possible further research stemming from this project. The author can definitely continue to test this algorithm with larger numbers of vertices per graph and different vertices-edges ratios. Another feasible contribution is to implement Parallel Edge-based Graph Coloring [1], which is expected to reduce runtime by a significant margin. A map to graph translation program can also be developed to replace random generated graphs with more practical graphs, which promise numerous future applications.

References

[1] Mehmet Deveci et al. “Parallel Graph Coloring for Manycore Architectures”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 892–901. DOI: `10.1109/IPDPS.2016.54`.

[2] Rudolf Fritsch et al. *Four-Color Theorem*. Springer, 1998.

[3] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*. John Wiley & Sons, 2011.

[4] David Kirk et al. “NVIDIA CUDA software and GPU parallel computing architecture”. In: *ISMM*. Vol. 7. 2007, pp. 103–104.