Graph Coloring using CUDA

I. Team Members

Name : André Vincent Pascal GROSSET
University ID : u0647404
Expertise : Computer Graphics and Visualization, First year PhD student working with Dr Charles Hansen

Name : Shusen Liu
University ID : u0647566
Expertise : Real-time Graphics, First Year PhD student.

Name : Peihong Zhu
University ID : u0609239
Expertise : Image Processing, Second Year PhD student.

II. Problem description

What is the Computation and Why it is important

Graph coloring is the assignment of labels or colors to to elements of a graph (vertices or edges) subject to certain constraints. In this project, we will consider how as few colors as possible can be assigned to vertices of a graph so that no two neighboring vertices (vertices connected by an edge) have the same color.

The convention of using colors comes from coloring countries on a map where each country should have a different color from its neighbor. However, countries on a map is an example of a planar graph and for planar graphs, four colors are enough. In the case of non-planar graphs, we do not know how many colors are required.

In graph theory, a planar graph is a graph that can be embedded in the plane, i.e, it can be drawn on the plane in such a way that its edges intersect only at their endpoints. An example of a map coloring (planar case) is shown in Figure 1 where neighboring states are colored using different colors.

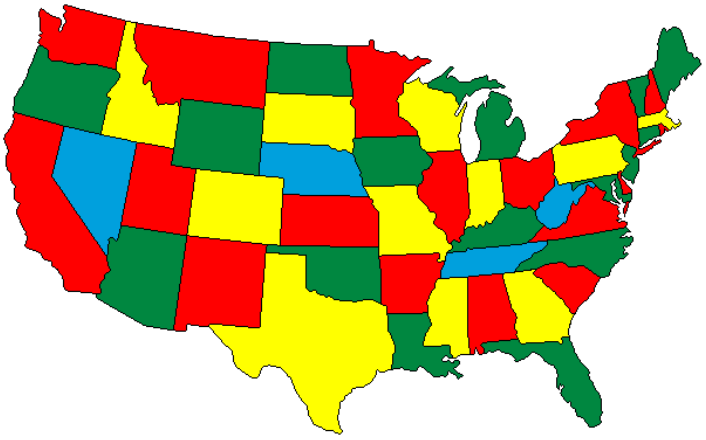

Figure 1 - Map coloring of the states in the U.S. : http://britton.disted.camosun.bc.ca/usamap/map_solution.gif

A more general example for a non-planar graph is shown in Figure 2.

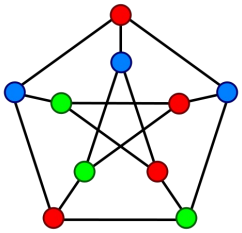

Figure 2 - Illustration of Graph Coloring, each vertex and different color with their adjacent ones : http://upload.wikimedia.org/wikipedia/commons/9/90/Petersen_graph_3-coloring.svg

The problem of graph coloring has several applications. For example, in assigning frequencies to wireless access points, we want to assign as few frequencies as possible and still do not want two access points have the same frequencies being neighbors of each other. Other examples include time-tabling and scheduling, optimization and printed circuit testing among others.

Abstraction of computation

For a graph G = (V,E), we denote the number of vertices |V| by n, the number of edges|E| by m, and the degree of a vertex vi by deg(vi). The maximum, minimum, and average degree in a graph are denoted by (delta), (sigma), and (sigma -) respectively. The minimum number of colors required to color a graph is referred to as the Chromatic number.

There exists several fast sequential coloring heuristics. They are based on the same general greedy framework: a vertex is selected according to some predefined criterion and then colored with the smallest valid color. The selection and coloring continues until all the vertices in the graph are colored.

For our project we are more interested in parallel version of these algorithms. Quite a few have been suggested in [1] but they fail to get any speedup as the number of processors increase. However, in [3] and [6] they do get speed up as the number of processors increase and so we are going to focus on that. The algorithm is shown below:

Algorithm details

GM-Algorithm from [6]

Phase 0 : Partition
Randomly partition V into p equal blocks V1 ..... Vp. Each processor Pi is responsible for coloring the vertices in block Vi.

Phase 1 : Pseudo-color

```
 for i = 1 to p do in parallel
   for each v 2 Vi do
     assign a legal color to v, paying attention
     to already colored vertices (both on and off processor).
```

Phase 2 : Detect conflicts
```
 for i = 1 to p do in parallel
  for each v in Vi do
   if any (v,w) in E s.t.color(v) = color(w) and v smaller and equal w
     Li  <- Li U {v}
```

Phase 3 : Resolve conflicts
Color the vertices in the conflict list sequentially

III. Suitability for GPU acceleration

The Amdahl's law states that "if P is the proportion of a program that can be made parallel, and $(1 - P)$ is the proportion that cannot be parallelized, then the maximum speedup that can be achieved by using N processors is"

$$\frac{1}{(1-P) + \frac{P}{N}}. \quad [7]$$

We would think that graph coloring would be a natural thing to do in a parallel way as we can split a big graph into several small regions and color each of them separately in parallel. This is exactly what most parallel graph coloring algorithm do. However, in [3] we see that as we increase the number of processors, the amount of conflicts also increase. We think and hope to be able to reduce the amount of conflicts by using better partitioning mechanisms and thus make it more suitable for GPU which compared to the work presented in [3] where they had up to 16 processors, GPU have hundreds of processors.

From the test results provided in [3], the algorithm can achieve a speedup up to 12.5x on a 16-node PC cluster equipped with dual 900 MHz Intel Itanium 2 CPUs and 4 GB memory. We believe this scheme can be implemented on GPU with CUDA, which has more powerful paralization computing capacity.

Synchronization and Communication

The first step of doing coloring in parallel is to partition the graph among the processors into interior and boundary vertices. An interior vertex is one of whose neighbors are located on the same processor as itself. And a boundary vertex has at least one neighbor located on a different processor. Clearly, the subgraph only contains interior vertices are independent of each other and hence can be colored concurrently trivially. Coloring the remainder of the graph in parallel requires communication and coordination among the processors and this is the main issue in the algorithm being described.

Since the subgraphs induced by interior vertices are independent of each other and can therefore be colored concurrently without any intra-processor communication, the interior vertices can be colored before, after or interleaved with boundary vertices.Coloring the interior vertices first may produce fewer conflicts when using a regular First-Fit coloring scheme, however, coloring boundary vertices first may be advantageous with color selection variants such as Staggered First-Fit scheme.

In practice, the supersteps can be made to run in a synchronous fashion by introducing explicit synchronization barriers at the end of each superstep. An advantage of this mode is that in the conflict detection step, the color of a boundary vertex needs to be checked only against its neighbors colored at the same superstep. The obvious disadvantage is that the barriers, in addition to the associated overhead, cause some processors to be idle while others complete their supersteps. Alternately, the supersteps can be made to run asynchronously, without explicit barriers at the end of each superstep. Each processor would then only process and use the color information that

has been completely received when it is checking for incoming messages.By doing so, in the conflict detection step, the color of a boundary vertex needs to be checked against all of its off-processor neighbors, and it is possible that the asynchronous version results in more conflicts than the synchronous one since a superstep on one processor now can overlap with more than one superstep on another processor.

Copy Overhead

Depending on the size of the graph (number of vertices) and the representation chosen (adjacency list or adjacency matrix) the amount of data to be transferred to the host and back will vary. The upper limit on the amount of data to be transferred is n*n*4 bytes (assuming the size of an integer is 4 bytes) to represent the graph and n*4 bytes to represent the color allocated to each vertex where n is the number of vertices.

Given that we plan to use a GTX 260 with 876 MB of Memory, we should easily be able to store data for a graph of 10 000 vertices in the global memory. Work with bigger graphs will involve working with different parts of the graph at each time on the GPU which is also feasible but will involve several transfers of data between host and device and consequently performance will suffer due to the numerous data transfers.

IV. Intellectual Challenges

All planar graphs can be colored using four colors [2]. However, for non-planar graphs, finding the minimum number of colors required to color a graph is an NP-Hard problem [4]. There is however an agreed lower limit which is the clique number for the graph; yet finding that for a graph is also an NP-complete problem for which brute-force algorithms are often used and so graph coloring algorithms do not usually try to determine the clique number.

For large graphs, operating simultaneously on independent parts of the graph should help to speed up the coloring process and this is exactly what parallel graph coloring algorithms exploit. Yet, here again, the focus is not always on finding the minimal number of colors for a graph but few enough colors as well as balancing the load between the different nodes doing the computation so that the coloring is as fast as possible.

Difficulties

One of the first difficulties we foresee is to get any speedup on the parallel algorithm! In[1], several algorithms are presented but there is no speedup when the number of processors increase as mentioned before, quite the contrary, performance declines. Eventually we will focus on the algorithm described in [5] that was continuously improved in [3] and [6].

The algorithm described has 4 phases and third phase is detection of collision which is then resolved in fourth phase. However, the amount of conflicts seem to increase with the number of processors as shown in [3]: in CUDA, compared to other platforms, we have many more cores. In the GTX 260, there are 192 cores and based on what we see in [3] where there are only 16 cores we foresee lots and lots of conflicts. Resolving these might take a heavy toll and we might have to modify the way the graph is initially partitioned to reduce the conflicts that eventually arise. Finding a good partitioning strategy will thus be critical.

Moreover, CUDA has a 512 limit on the number of threads in a block. Consequently for large graphs, computation might have to spread among several blocks. This might pose a problem as interblock communication is not that obvious in CUDA. Moreover, depending on the number of nodes in the graph and the algorithm that is chosen, extensive communication might be required.

V. Conclusion

Quickly coloring a graph with the minimal number of colors is a hard problem. For this project, we will try to get the best trade-off between the minimum number of colors used and maximum speed. Moreover, we have not found any implementation of graph coloring using CUDA, thus this project will be very interesting project especially if we manage to get decent speed ups.

VI. References

[1] J.R. Allwright, R. Bordawekar, P.D. Coddington, K. Dincer, and C.L. Martin, "A Comparison of Parallel Graph Coloring Algorithms," 1995.

[2] K. Appel and W. Haken, "Every Planar Map is Four-Colorable, II: Reducibility." *Illinois J. Math.* 21, 491-567, 1977.

[3] E.G. Boman, D. Bozda, U. Catalyurek, A.H. Gebremedhin, and F. Manne, "A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers," *Work*, pp. 1-10.

[4] M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W. H. Freeman & Co. New York, 1979

[5] A.H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, 2000, pp. 1131-1146.

[6] A.H. Gebremedhin, F. Manne, and T. Woods, "Speeding up Parallel Graph Coloring," *Optimization*, pp. 1-10.

[7] Wikipedia: Amdahl's Law [Online] Available: http://en.wikipedia.org/wiki/Amdahl%27s_law

[8] Wikipedia: Graph Coloring [Online] Available:http://en.wikipedia.org/wiki/Graph_coloring