Parallel Graph Coloring Algorithms on the GPU Using OpenCL

Shilpi Sengupta
Dept. of Computer Science & Engineering,
JSSATE
Noida, India
shilpi.sengupta@gmail.com

***Abstract*— GPUs (Graphics Processing Units) are designed to solve large data-parallel problems encountered in the fields of image processing, scene rendering, video playback, and gaming. GPUs are therefore designed to handle a higher degree of parallelism as compared to conventional CPUs. GPGPU (General Purpose computing on Graphics Processing Units) enables users to do parallel computing on the graphics hardware commonly available on current personal computers. These days' systems are available with multi-core GPUs that provide the necessary hardware infrastructure, thereby enabling high performance computing on personal computers. NVIDIA's CUDA (Compute Unified Device Architecture) and the industry standard OpenCL (Open Computing Language) provides the software platform required to utilize the graphics hardware to solve computational problems using parallel algorithms, otherwise solvable mostly in supercomputing environments. This paper presents two parallel CREW (Concurrent Read Exclusive Write) PRAM algorithms for optimal coloring of general graphs on stream processing architectures such as the GPU. The algorithms are implemented using OpenCL. The first algorithm presents the techniques for computing vertex independent sets on the GPU and then assigns colors to them. The second algorithm focuses on the optimization of the vertex independent set computation for edge-transitive graphs by taking advantage of the structures of such graphs and then assigns color to each of the normalized independent sets.**

Keywords— graph; GPU; OpenCL; vertex color; VIS

I. INTRODUCTION

Graph based problems are the appropriate choices for data parallel computing. In this paper, we present data parallel graph algorithms for vertex coloring. These algorithms are implemented on NVIDIA [1] graphics card using OpenCL (Open Computing Language)[2]. High-end graphics cards that are now-a-days available in personal computers provide researchers with a conveniently-available, high-performance parallel programming environment. Graph (vertex) coloring problem is a well-known NP Complete problem in the area of discrete mathematics. Graph coloring problem finds applications in diverse fields. Some of these areas include puzzle-solving (Sudoku) [3], Scheduling cell transmission [4], time-tabling [5], Register Allocation [6], Map coloring, Frequency Assignment Problems [7], Biological Networks (Protein-protein interaction networks) [8]. Such a wide range of application areas make graph coloring a very intriguing and sought after area of research [9].

In this paper, we present a vertex coloring algorithm that colors an undirected graph by partitioning it into vertex oriented independent sets (VIS). For a given graph *G* with n vertices and maximum vertex degree Δ, our algorithm finds a proper κ-coloring of the vertices of *G* with κ colors where κ is less than or equal to Δ + 1. We also present a specialized version of this algorithm for edge-transitive graphs where the graph is colored using κ colors; κ is equal to the chromatic number of the graph.

II. PRELIMINARIES

A graph denoted is as G = {V, E}, where V is the set of all vertices in G and E is the set of all edges in G. The i^{th} vertex of the graph G is represented as v_i, and the edge connecting vertices v_j and v_k is represented by $e_{j,k}$. |E| represents the number of edges in the graph, and |V| represents the number of vertices in the graph. The completeness factor C (or connectedness) of graph denotes how connected the vertices are and is represented by a real number. The C of a complete graph is 1.0. The C of an undirected graph can be computed using the following formula:

Number of edges in a complete graph G of n vertices:

$$|E|_{COMPLETE} = {}^{n}C_2 = n\,(n-1)\,/\,2 \qquad (I)$$

For a graph G with n vertices and |E| edges, the completeness factor C is defined as:

$$C = |E| / |E|_{COMPLETE} \qquad (II)$$

For a graph G with n vertices and |E| edges, the storage complexity of the degree sorted adjacency list of the complement G' is: ${}^{n}C_2 - |E| + n$. An Edge Transitive graph [11] G is a graph such that for any two edges e_1 and e_2 in G the vertex label and (or) the edge labels are the only way of distinguishing e_1 from e_2. If these labels are removed then there does exist any characteristic feature of the edges, such as connectivity structure of its end vertices that can uniquely

978-93-80544-12-0/14/$31.00

identify e_1 from e_2. Example of such graphs include $K_{5,5}$, Hamming Graph, Cycle C_{15}.

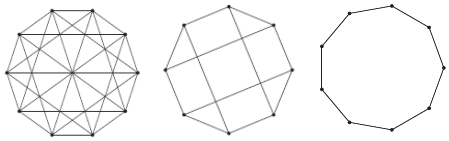

Fig 1. Edge Transitive Graphs, K5,5, Hamming, Nonagon

A **V**ertex **I**ndependent **S**et vis_i of a vertex v_i of graph G is a collection of vertices that are not adjacent to v_i, and are not connected to each other. Formally, in graph G = {V, E}, if $e_{r,q}$ denotes an edge in G from vertex v_r to v_q, where V denotes the set of all vertices in G and E denotes the set of all edges in G, then: vis_i = { v_j, v_k □ G; ∃ v_i □ V, $e_{j,k}$ ∉ E and $e_{i,k}$ ∉ E and $e_{i,j}$ ∉ E }; thus, each vertex of the graph G has a VIS. Calculating the VIS for each vertex is an inherently parallel operation and does not require any inter-processor communication. The experimental results show that in symmetric graphs, VIS for multiple vertices are identical. In such graphs the first few distinguishable VIS include all the vertices of the graph, hence it is not required to iterate through all the VIS during the color assignment phase. In the algorithms we present in this paper, the work of partitioning the graph by finding VIS is performed in parallel on the GPU threads. Each thread is assigned a vertex for VIS computation.

An adjacency bit-matrix was adopted for representing the adjacency matrix of a given input graph. Furthermore this representation was optimized to store only the upper right triangle of the matrix thus requiring $n^2/16$ bytes. The principle of locality-of-reference played a key role in the reducing the data access speeds on both the CPU and the GPU thereby minimizing cache misses.

OpenCL [12] is the open standard for parallel programming of heterogeneous systems created by Khronos [2] group. Prior to the advent of languages as OpenCL and CUDA [1], the use of graphics processor units was confined only to the field of high end graphics computations. The diverse parallel architectures that can be programmed using OpenCL include GPUs, multi-core CPUs, DSPs and more. OpenCL 1.0 supports data parallel model.

III. ALGORITHMS

The proposed algorithm is divided into three phases, namely, VIS Generation phase, VIS Coloring phase and Conflict Resolution phase. The first phase is completely parallel in nature and does not call for any inter-processor communication. VIS Coloring phase entails synchronization among each kernel executing for each vertex. The vertices for which the color is not decided during the second phase are assigned color sequentially in the Conflict Resolution phase. We now formally describe the steps of the algorithm. This is followed by a small example demonstrating the VIS generation of the vertices and the colors assigned to the vertices by the algorithm.

Table 1: Algorithm GRAPH_COLOR

Algorithm GRAPH_COLOR

Set the adjacency matrix, colorVertexArray[] in the shared memory of the GPU

Launch the kernel PARALLEL_VERTEX_COLOR , for ∀ vertices j□ adjacency[i,j]
Read back the colorVertexArray[] array from the GPU
Call function **CONFLICT_RESOLUTION()**
End GRAPH _COLOR

Table 2: Algorithm PARALLEL_VERTEX_COLOR

Input: adjacency bit matrix of the graph

Output: colors assigned to vertices in the colorVertexArray[]

Kernel Shared Data: adjacency[], numVertices, colorVertexArray[], groupVIS[], minimumAvailableColor. These data structures are shared amongst all executing kernels on the GPU.

Notes: This algorithm is a stream kernel. It is called in parallel for all the vertices v = 1, 2, ... |V|. This algorithm uses CAS (Compare and Swap) implemented on the GPU instruction set.

```
function PARALLEL_VERTEX_COLOR(v) [KERNEL]
begin
nonAdjacency[v] ← CALCULATE_NON_ADJACENCY (v)
  for∀ i ∈ nonAdjacency[v] do
    for∀ j □ groupVIS[v] do
          select j in order of degree(j) such that degree(j) ≤
degree(i)
          if adjacency [i,j] is false
              groupVIS[v] ← groupVIS[v] ∪ i
          endif
     endfor

// Color the vertex on the basis of the VIS
     ifnot IMPURE( groupVIS[v] )
      for∀ i □ groupVIS[v] do
        CAS( colorVertexArray[i], minimumAvailableColor )
      end for
     CAS( minimumAvailableColor, minimumAvailableColor
+ 1 )
   end if
  end for
end PARALLEL_VERTEX_COLOR
```

Table 3: Algorithm CALCULATE_NON_ADJACENCY

Input: graph vertices
Output: the non adjacency list is calculated for the vertices
Kernel Shared Data: nonAdjacency[]
Notes: This algorithm is a stream local function. It is

called in parallel for all the vertices v = 1, 2, ... |V| from the stream kernel PARALLEL_VERTEX_COLOR

```
algorithm CALCULATE_NON_ADJACENCY(v) [LOCAL]
begin
  initialize nonAdjacency[0]← 0
  // The first element specifies the length of the nonAdjacency list for vertex v
  for∀ vertices i← 1,2,...|V| do
    if adjacency[v,i] is false
      nonAdjacency [0] ← nonAdjacency [0] + 1
        for∀ j □ nonAdjacency [v]
            add i in nonAdjacency [v] at the location such that degree[v]≥ degree[j]
                end for
          end if
    end for
end CALCULATE_NON_ADJACENCY
```

Table 4: Algorithm IMPURE

Input: group VIS for vertex v
Output: returns true if the groupVIS is impure otherwise false. If the group VIS contains a vertex that is already colored, then the groupVIS is impure.
Kernel Shared Data: colorVertexArray[]
Notes: This algorithm is a stream local function. It is called in parallel for all the vertices v = 1, 2, ... |V| from the stream kernel PARALLEL_VERTEX_COLOR

```
algorithm IMPURE( groupVIS[v] )
begin
  for∀ i □ groupVIS[v] do
     if colorVertexArray[i] != -1
        return true
     end if
  end for
end IMPURE
```

Table 5: Algorithm CONFLICT_RESOLUTION

Input: graph vertices
Output: returns true if the groupVIS is impure otherwise false. If the group VIS contains a vertex that is already colored, then the groupVIS is impure.
Notes: This algorithm is executed sequentially on the CPU for all vertices that have not been colored previously. Once the stream kernels have finished execution, the colorVertexArray stream data is copied to the CPU memory.

```
algorithm CONFLICT_RESOLUTION() [CPU – SEQUENTIAL]
for∀ vertices i← 1,2,...num-vertices do
        if not colored( colorVertexArray [i] )
                // j range between 0 and degree[i]
                for∀ j □ adjacency[i,j] do
                        colorVertexArray[i] ← first available color
                end for
        end if
end for
end CONFLICT_RESOLUTION
```

A. Application of the algorithm to the Petersen Graph

We now perform the algorithm PARALLEL_VERTEX_COLOR on the Petersen graph. Petersen Graph is an undirected graph with 10 vertices and 15 edges. The chromatic number of the graph is 3.

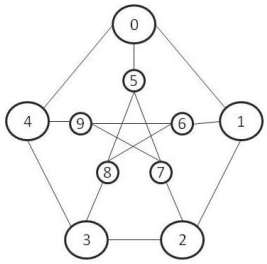

Fig 2. Petersen Graph

Using the PARALLEL_VERTEX_COLOR algorithm, the graph is colored with proper κ coloring where κ = 3 colors. The results are shown in the following table.

Table 6: Petersen Graph Results

Vertex v	groupVIS[v]	Color[v]
0	0 2 6	Red
1	1 3 5 9	Yellow
2	0 2 6	Red
3	0 3 6 7	Yellow
4	1 4 5	Green
5	1 3 5 9	Yellow
6	0 2 6	Red
7	0 3 6 7	Green
8	0 2 8 9	Green
9	0 2 8 9	Yellow

B. Parallel Vertex Coloring for Edge Transitive Graphs

In the following section we present a modified version of algorithm PARALLEL_VERTEX_COLOR for coloring edge-transitive graphs. This algorithm performs graph coloring for edge-transitive graphs in lesser time by reducing the number of iterations in the CONFLICT_RESOLUTION phase.

Table 7: Algorithm GRAPH_COLOR

```
algorithm CONFLICT_RESOLUTION_FOR_EDGE_TRANSITIVE() [CPU – SEQUENTIAL]
        for every uncolored vertex v do
```

```
            for∀ j □ groupVIS[v]do
                  maxColor = max(colorVertexArray[j])
                  colorVertexArray[i] ← maxColor + 1
            end for
      end for
end
CONFLICT_RESOLUTION_FOR_EDGE_TRANSITIVE
```

IV. COMPLEXITY

A. VIS Generation

The complexity of VIS generation for each vertex v_i in graph is:

$$(|V| - d(v_i)) \times (|V| - d(v_i) -1) / 2$$

The upper bound of this function is $(|V| - d(v_i))^2$ per processor.

B. Impurity Test

To test the impurity of a VIS, the algorithm IMPURE will loop a maximum of $|V| - d(v_i)$ times per processor.

V. EXPERIMENTAL RESULTS

A. Setup

The algorithm was tested on the following configurations:

a. Microsoft Windows Vista SP2 Home Premium on Intel Core2 Duo CPU T5500 @ 2.20 GHz each, 3070 MB RAM, with NVIDIA GeForce 8400M GPU with 128MB of dedicated graphics memory (16 cores)
b. Microsoft Windows Vista Enterprise on Intel Core2 Duo CPU T7700 @ 2.40GHz each, 3070 MB RAM, with NVIDA Quadro FX 570M GPU with 512 MB of dedicated graphics memory (32 cores).

B. Graph Generator

To test our algorithm, a graph generator was developed to generate random graphs for creating input data set for our algorithm. The graph generator created random graphs based on the number of vertices and the completeness factor as inputs. Apart from these generated graphs several well-known graphs from previous experimental setups were tested on the new algorithm.

C. Results

Table 8: Results of PARALLEL_VERTEX_COLOR_FOR_EDGE_TRANSITIVE on Well Known Graphs

Data Set	|V|	|E|	Colors
Tetrahedral	4	6	4
Octahedral	6	12	3
6-cycle	6	6	2
Utility	6	9	2
34bipartite	7	12	2
7- star	7	6	2
Cubical	8	12	2
Peterson	10	15	3
Cuboctahedral	12	24	3
Heawood	14	21	2
Dodecahedron	20	30	3
$K_{3,3,3}$	9	27	3

The above experimental results for symmetric graphs prove the correctness of algorithm and in these cases the number of the colors used is same as the chromatic number of the graph.

VI. CONCLUSION

In this paper we presented parallel algorithms for generating near optimal coloring of random graphs and optimal coloring for edge-transitive graphs using GPUs commonly available on current desktop systems. These algorithms were implemented on the GPU using the industry standard OpenCL language. We showed that VIS computation is inherently a data parallel problem and is efficiently handled on stream processing architectures like the GPU. We then applied the VIS data to compute groups of vertices that can be colored independently of each other. The computed groups were then colored on the CPU. We saw using GPU resources, parallel algorithms can be applied to solve problems that could hitherto be solvable only on high-end computing systems.

REFERENCES

[1] NVIDIA, http://www.nvidia.com, 22-10-2013

[2] OpenCL (Open Computing Language), http://www.khronos.org/opencl/, 22-10-2013

[3] Agnes M. Herzberg and M. Ram Murty, "Sudoku Squares and Chromatic Polynomials", Notices of the American Mathematical Society, http://www.ams.org/notices/200706/

[4] Lakshman, T.V.Bagchi, A.Rastani, K.Bellcore, "A graph-coloring scheme for scheduling cell transmissions and itsphotonic implementation", IEEE Transactions on Communications,1994

[5] F.T. Leighton, "A graph coloring algorithm for large scheduling problems",Journal of Reasearch of the National Bureau of Standards, 84:489-506, 1979.

[6] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation", ASCM Conference on Program Language Design and Implementation,1989

[7] Andreas Gamst, "Some lower bounds for a class of frequency assignment problems", IEEE Transactions of Vehicular Echnology, 35(1):8-14, 1986.

[8] Khor Susan, CERN Document Server: Record#1229603: Application of Graph Coloring to Biological Networks, http://cdsweb.cern.ch/record/1229603

[9] M.Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem", Proceedings of the seventeenth annual ACM symposium on Theory of computing,1986

[10] John D. Valois, "Lock-Free Linked Lists Using Compare-and-Swap",Fourteenth Annual ACM Symposium on Principles of Distributed Computing, 1995

[11] ME Watkins, "Connectivity of transitive graphs - Journal of Combinatorial Theory",Elsevier,1970

[12] www.nvidia.com/content/cudazone/download/OpenCL_ProgrammingGuide.pdf, 22/10/2013