Analysis of Parallel Incremental/Decremental Graph Colouring on GPU

A Project Report

submitted by

MOHAMMED SHAMIL

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

under the guidance of

Dr. Rupesh Nasre

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

MAY 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **Analysis of Parallel Incremental/Decremental Graph Colouring on GPU**, submitted by **Mohammed Shamil**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Rupesh Nasre
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT Madras, 600 036

Place: Chennai

Date: 11 May, 2016

ACKNOWLEDGEMENTS

Thanks to all those who made TEX and LATEX what it is today.

ABSTRACT

KEYWORDS: Colour Quality; Compressed Sparse Row Representation; Decremental Graph Colouring; GPGPU; Graph Colouring; Incremental Graph Colouring; NP-hard; nVIDIA Cuda; Parallel Computing; Parallel Graph Algorithms; Vertex Colouring.

Graphs are a well studied and widely used data structure in the field of algorithms, programming and computing. There are a lot of interesting applications of graphs and various algorithms are built on top of the graph data structure. One of the most famous and well studied graph problems is that of graph colouring. There are a lot of different versions of graph colouring problem of which the most common ones are that of vertex colouring and edge colouring. The problem is seemingly simple, to allocate a colour to every vertex/edge of a graph so that adjacent vertices/edges don't share the same colour minimizing the number of colours used. Graph colouring is a very important and yet very challenging graph problem with ongoing active research. Graph colouring finds application in a varied range of problems including various scheduling problems like job scheduling on distributed computing systems, register allocation in compilers, pattern matching problems and solving Sudoku boards.

Though the problem is seemingly simple, it is computationally hard. The graph colouring problem we are exploring in this work, that of vertex colouring, is an NP-hard problem. The sequential approaches like greedy colouring are simply not fast enough whereas advanced approximate/randomized solutions either produce colourings of bad colour quality or aren't fast enough. Thus came the parallel approaches to Graph Colouring. Most of the parallel versions of Graph Colouring algorithms were designed with either multi-core CPUs or heavy duty super computers in mind. With the advent of General-Purpose computing on GPUs (GPGPU), we have access to cheap heavy multi-threaded parallel computing power. Our work is based on parallel computing on nVidia GPUs using Cuda programming language.

We explore different parallel graph colouring algorithms on nVIdia GPUs in this work and try to adapt them to support addition of edges, called incremental graph colouring, and deletion of edges, called decremental graph colouring. In the first section, we explore different parallel graph algorithms and adapt a couple of them, one based on *speculation* and *conflict resolution* and the other on *Vertex Independent Sets*, to work on nVidia GPUs. In the following sections, we adapt the GPU parallel colouring algorithm to support additions and deletions of edges. In the incremental part, we explore different methods to maximize parallelization while colouring newly added edges and use propagation to improve overall colour quality. In the decremental part, we explore different options to either process the vertices, on which the deleted edges were incident, on the go or to process them together and use propagation to propagate the information across the graph improving the colour quality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
RTFM	Read the Fine Manual
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Units
CSR	Compressed Sparse Row

CHAPTER 1

INTRODUCTION

1.1 Graphs and Graph Algorithms

Graphs are really important mathematical concepts and in the area of computing, their various forms are widely used as data structures to aid various algorithms. Graphs are commonly used to denote relations between different entities and hence is a very important and integral part of many algorithms. On a practical level, we deal with graphs in the order of billions of nodes and edges on a daily basis. Especially with the advent of social networks and big data, a lot of active research is ongoing in the analysis and understanding of large graphs.

Many problems in the area of Computer Science, Biology etc. are solved with the help of algorithms which are based on graphs. Shortest path problem, Travelling Salesman Problem (TSP), network flow problems, vertex cover problem, graph colouring etc. are important graph-based problems with many practical applications in the real world. Our work is on Graph Colouring which is one of the most famous and well studied graph problems.

1.2 Vertex Colouring

Graph Colouring problem entails *colouring/labeling* of the vertices/edges of a graph based on some set of conditions which are to be satisfied. In other words, its a problem in which you allocate a colour/number to every vertex/edge of a graph such that a set of constraints are satisfied. There are different versions of Graph Colouring and the one which is of interest to us is that of Vertex Colouring.

1.2.1 Classical Vertex Colouring Problem

Vertex Colouring is the most basic version of Graph Colouring and other Graph Colouring problems can be presented as a Vertex Colouring problem. In it's classical form, Vertex Colouring is:

> ***Vertex Colouring:*** *Colouring all the vertices of a graph such that adjacent vertices have different colours. That is, there shouldn't be an edge where the incident vertices share the same colour.*

There are other forms of vertex colouring where additional conditions than the one given above need to be considered while colouring. In our work, we are concerned only with the classical form of vertex colouring which hereinafter interchangeably referred to simply as Graph Colouring.

1.2.2 Chromatic Number $\chi(G)$

A graph *G* is said to *k*-colourable, if *G* can be coloured using *k* colours. For example, from the *Four Colour Theorem*, we have that all planar graphs are 4-colourable. Also, all bipartite graphs are 2-colourable.

The *Chromatic Number* of a graph G, denoted by $\chi(G)$, is the minimum number of colours required to colour a graph. That is, $\chi(G)$, is the minimum value of all *k* for which the graph *G* is *k*-colourable. Therefore, if a graph is *k*-colourable, we have:

$$\chi(G) \leq k$$

1.2.3 Colour Quality

Colour Quality is a term used to denote how good the colouring done by a particular algorithm is. Colour Quality is said to be better for an algorithm if the number of colours used by the algorithm to colour a graph *G* is closer to its Chromatic Number, $\chi(G)$.

Mathematically, Colour Quality of a colouring is said to be better as the fraction,

$$\frac{No.\ of\ colours\ used\ by\ the\ algorithm}{\chi(G)}$$

is closer to 1.

1.2.4 Complexity

Graph Colouring is a computationally complex problem. To decide if a Graph can be coloured using *k* colours, is an NP-complete problem. Whereas, finding the Chromatic Number of a graph ($\chi(G)$) is proved to be an NP-hard problem.

There exist many algorithms like Greedy Colouring, approximation algorithms and randomized algorithms. There also exist polynomial time algorithms for some specific family of graphs. For example, it can be decided if a graph can be coloured using 2 colours by checking if it is a bipartite graph. This can be done in polynomial time using Breadth First Search (BFS).

1.2.5 Applications

Graph Colouring problem, which started as a map colouring problem (four colour theorem), finds many important real applications including but not limited to:

- Scheduling problems like job scheduling across multiple nodes in a distributed computing environment
- Register allocation problem during compilation
- Solving Sudoku
- Pattern matching applications

1.3 Parallelization

1.3.1 Frequency Scaling

Moore's law, which observes that the number of transistors present in an integrated circuit approximately doubles every two years, still stands valid. Processors, and hence

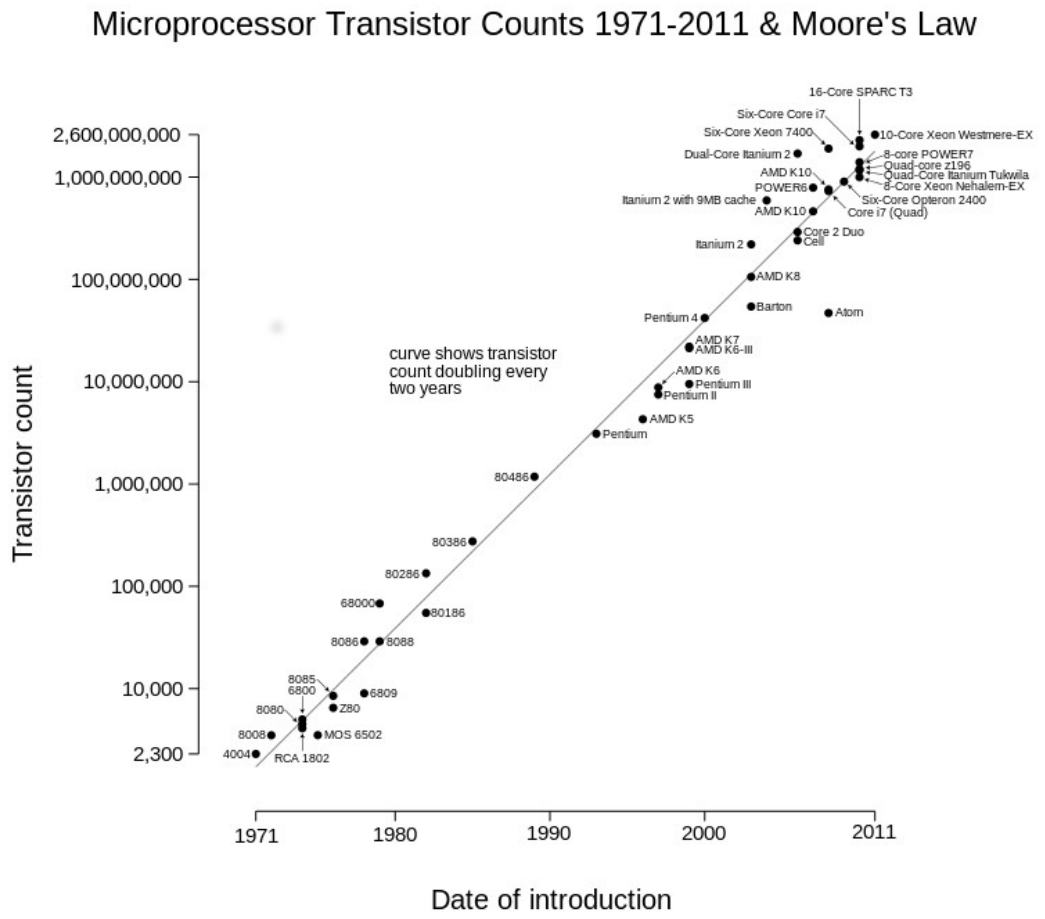

Figure 1.1: Graph showing Moore's Law in action, from Wikipedia, the free encyclopedia (2016). Each data point is a processor.

computers, have grown faster and faster over years. More and more transistors meant the processors could run faster, at a faster frequency. Processors with better and better clock speeds were introduced every year since the 1980s until around 2004 when instead of single core processors running at faster clock speeds, multi-core processors started rolling out.

1.3.2 Why Parallelization?

Around 2004, Intel and other processor manufacturing companies came to realize that frequency scaling was not practical any more. The increase in frequency meant an increase in power consumption which in turn meant an increase in heat generation. Thus it was no longer practical to increase the clock speeds of processors. Rather,

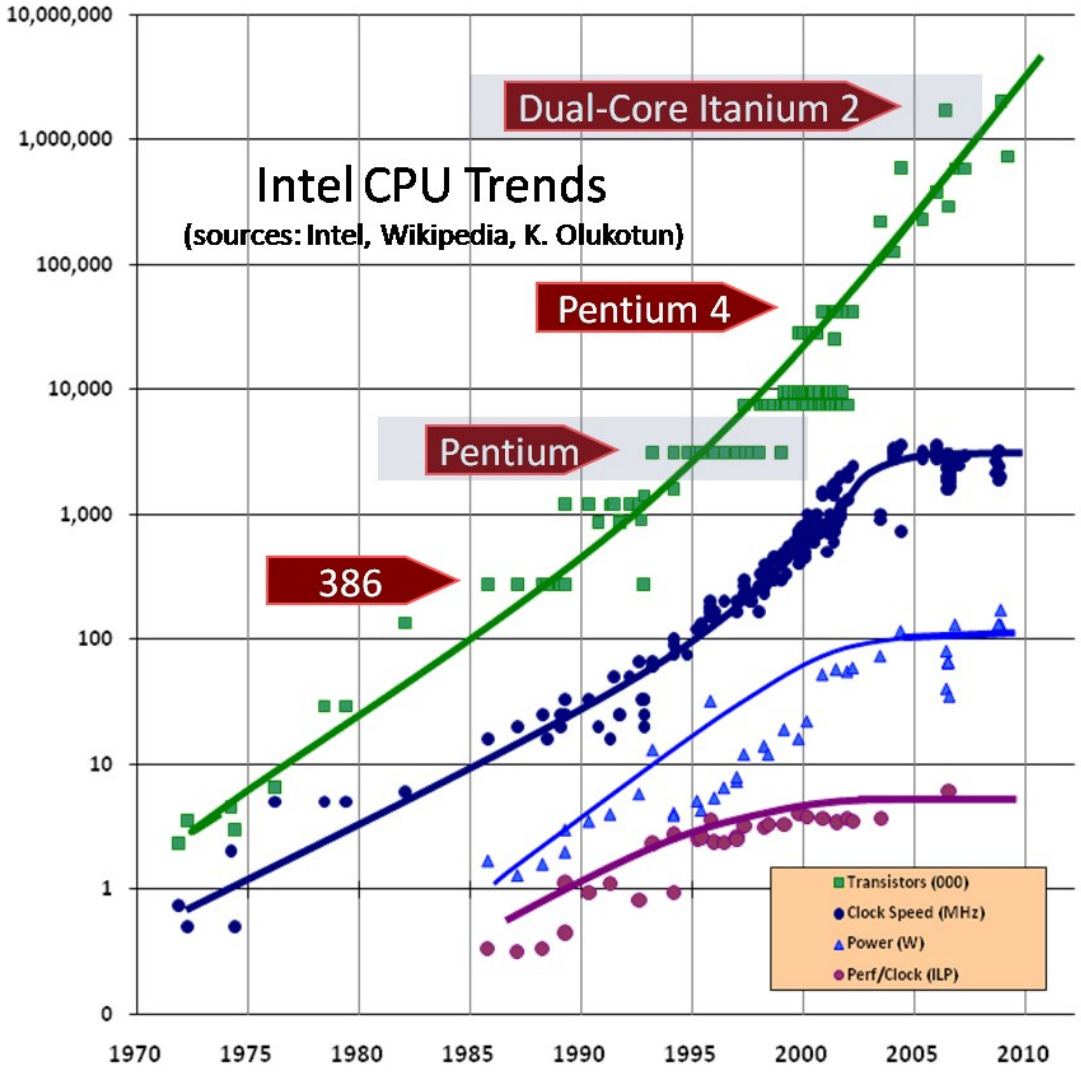

Figure 1.2: *Intel*'s transition from single core processors to multi-core processors around 2004-2005, from Sutter (2005).

they started making processors with the same clock speeds, but with multiple cores. Since then the computer architecture industry held fast to the paradigm of multi-core processors. This, in the case of *Intel*, is indicated in the figure 1.2.

Parallelization enables us to run programs faster by splitting the work across different cores of a processor which are ideally run in parallel. In an ideal setup, with n cores, we should see a speed up of n, which means the running time will become $1/n^{th}$ compared to the running time when run on a single core processor. Though we never really reach this ideal speedup, as stated by Amdahl's law etc., we still achieve significant speed ups.

1.3.3 Parallezation of Graph Colouring

Since the computer architecture industry made a shift to the multi-core paradigm, there had to be a shift in programming paradigm to support the newly available parallelism. Almost all the algorithms, programs etc. were designed and developed to run sequentially on a single core processor. Things have changed recently as more and more algorithms and programs are redesigned and redeveloped to make use of the newly available parallel hardware.

As discussed earlier, Graph Colouring is a computationally complex problem. It is NP-hard to solve. Also, the approximation algorithms for colouring a graph with n vertices are also NP-hard within $n^{1-\epsilon}$ for all $\epsilon > 0$. The existing solutions are either slow or are fast but produce bad colour quality. Also, practical graphs these days are very large with billions of vertices and edges. So, since the advent of parallel programming paradigms, there have been efforts to parallelize this well celebrated graph problem though most of them were meant specifically for distributed computing setups and super computers. In our work, we focus on parallel graph colouring which can be run on parallel hardware available locally. Especially with the advent of GPGPUs, cheap massive parallelism is at a hand's reach.

1.4 GPGPU

In the domain of parallel programs and applications, one big deterrent was that the number of processor cores available for parallelism was small. Most of the multi-core processors have 32 cores at the maximum. Only super computers had a very high number of cores and they came at a price.

1.4.1 Why GPUs?

Graphics Processing Units, GPUs, have been using parallelism since their birth. They have almost always been very accessible to the normal public as they are much cheaper than super computers. They also came with thousands of cores. But they were spe-

cialized for graphics related operations. Then came the paradigm of GPGPU, General Purpose computing on Graphics Processing Units. And with that, it was now possible to run regular operations and not just graphics related operations on the GPU. GPGPU brought with it easy, cheap access to massive parallelism.

1.4.2 nVidia CUDA

nVidia, one of the biggest players in the GPUs market, introduced its famous parallel computing platform, CUDA, in 2006, thus enabling easy GPU based parallel acceleration. In our work, we use CUDA C to parallelize graph colouring. CUDA lets us harness the power of thousands of cores in the CUDA enabled nVidia GPUs.

Architecture

In an nVidia GPU, as shown in the figure 1.3, there are multiple streaming multiprocessors, SMs, and each of these multi processors have thousands of cores/processors in it. Functions to be executed on the GPU are called Kernels and Kernels, once invoked, spawn the required number of threads as blocks of threads which are then executed across SMs. All threads in a block have access to the shared memory inside the SM in which that block is executed.

1.4.3 Challenges

Though, GPUs let us access thousands of threads easily, it comes with a cost. GPU memory and CPU memory are mutually exclusive. So, we have to first copy all the data which are to be processed by the GPU threads to the GPU before invoking the kernels. As the communication between the CPU and the GPU is enabled through the PCI-Express port, which is not super fast, there is a cost for transferring data between the CPU and the GPU, the so called *memory latency*. This data transferring cost is one of the biggest overheads in GPU computing. So, as a GPU programmer, one must try to reduce data transfer between the CPU and the GPU.

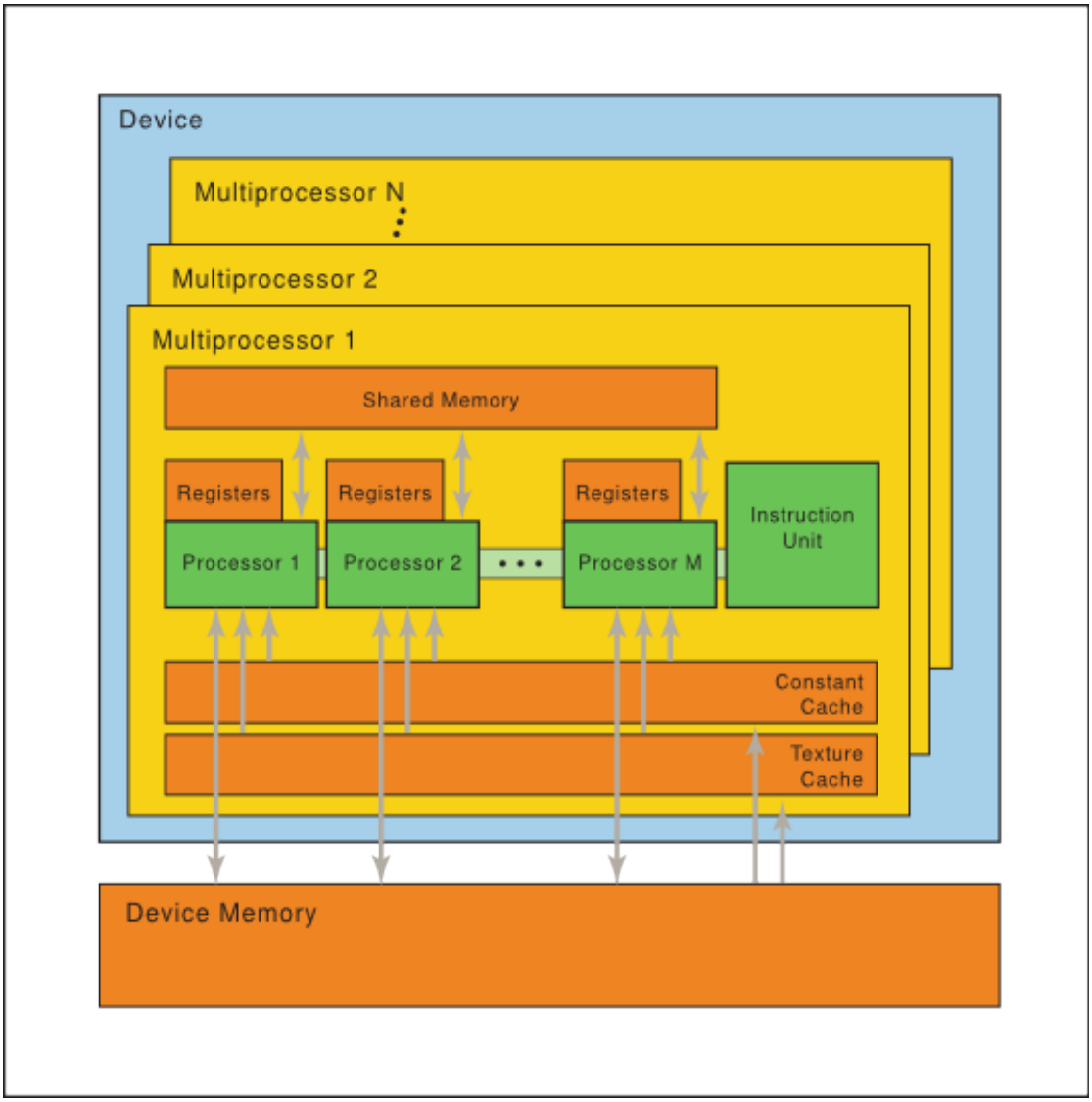

Figure 1.3: nVidia GPU hardware model, from NVIDIA Corporation (2016).

1.5 Incremental/Decremental

As we discussed already, Graph Colouring is a very important algorithm on graphs. We have so many practical applications for the same. But most of the practical graphs are dynamic in nature. Vertices and edges are added and deleted often. But the number of these changes are very small compared to the size of the full graph. So, it follows that it is not wise to rerun the graph colouring algorithm on the entire graph every time some vertices/edges are added or deleted. Our work on Incremental/Decremental graph colouring tries to take care of precisely the same.

In our work, we try to re-colour only the relevant parts of the graphs on addi-

tion/deletion of edges instead of re-colouring the entire graph. We consider only addition/deletion of edges as deletion of a vertex is considered as the deletion of all edges incident on that vertex. We consider different implications of Incremental/Decremental colouring such as the amount of time we save versus maintaining/improving the colour quality.

CHAPTER 2

PARALLEL GRAPH COLOURING

We have established why we are interested in the paradigm of parallel programming and why we want to parallelize graph colouring.

2.1 Graph Colouring Problem

The problem we are concerned with is that of *1-distance Vertex Colouring* or simply, *Vertex Colouring*.

> ***Parallel 1-distance Vertex Colouring:*** *Colouring all V vertices of a graph G(V,E) in parallel, such that vertices at a distance of 1 edge, adjacent vertices, don't share the same colour.*

2.2 Related Work

Graph Colouring is a well studied problem and there have been so many works on the same over years. But most of them were regarding sequentially solving the problem using various paradigms, like that of semi-definite programming, integer programming etc.

Recently there have been some parallel approaches to the same, but most of them like the works by ÇAtalyüRek *et al.* (2012) are based on Super Computers or other expensive hardware. There have been only a few work done regarding parallel graph colouring on GPUs like Grosset *et al.* (2011), Naumov *et al.* (2015) and Sengupta (2014) and none regarding Incremental/Decremental versions as far as the author understands.

2.3 Broad Classification of Parallel Graph Colouring Algorithms

As we are dealing with NVIDIA GPUs, we are concerning ourselves with algorithms pertaining to shared memory architectures only. Most of the parallel colouring algorithms based on shared memory architecture can be broadly classified into two of the following categories:

2.3.1 Vertex Independent Sets and Colouring

Algorithms falling under this category work in two phases:

1. Vertex Independent Sets: Find VIS
2. Colouring: Colour the VIS found without conflicts

This set of algorithms relies upon finding Vertex Independent Sets of vertices and colouring them in parallel. Most of the earlier parallel algorithms developed for graph colouring were based on this idea.

Vertex Independent Set (VIS): A vertex independent set of a graph G is a set of vertices who don't have any edges amongst each other.

Mathematically, γ is a valid Vertex Independent Set of a graph $G(V, E)$ if

$$\gamma \subseteq V$$

and

$\forall\, v_i, v_j \in \gamma$, $e \in E$, if v_i is incident on e, then $\forall\, j \neq i$, v_j is not incident on e.

The idea is pretty straight forward. These are iterative algorithms where, in each iteration, you find a Vertex Independent Set of the given graph and colour all of the vertices in that VIS with a single colour. The process is continued with different colours until there are no more vertices to be coloured. Both the steps, finding VIS and colouring the vertices in that VIS, can be done in parallel. This category of algorithms roughly work as explained in Algorithm 1.

Algorithm 1 Vertex Independent Sets and Colouring

```
1: procedure VISParallelGraphColouring(G(V,E))
2:     Initialization                ▷ Initialize all the variables and other data structures
3:     currentColour ← 1
4:     while V ≠ φ do                ▷ Run until all the vertices are coloured
5:         γ ← a VIS of G(V,E)       ▷ VIS can be found in parallel
6:         for each v ∈ γ do         ▷ This loop can be run in parallel
7:             colour[v] ← currentColour
8:         V ← V − γ
9:         currentColour ← currentColour + 1
```

2.3.2 Speculation and Conflict Resolution

Algorithms falling under this category work in two phases:

1. Speculation: Colour the graph based on some pre-existing knowledge possibly generating conflicts
2. Conflict Resolution: Resolve the conflicts possibly generated in the first phase

The first category of algorithms relied upon finding Vertex Independent Sets iteratively so that we can colour the vertices in the VIS found in each step without any conflict. This second category of algorithms instead let us commit some mistakes, or rather conflicts, in our colouring. That is, it saves us from finding a VIS in each step, instead we colour the graph using some pre-existing knowledge like existing colouring of the graph or some structural information regarding the graph.

So, in the first phase, instead of finding a VIS and colouring just the vertices in that VIS without any conflict in an iteration, we speculate the colours of the entire graph with some pre-existing knowledge and possibly commit conflicts. The possible conflicts inflicted in this first phase are rectified in the second phase in which we find the conflicts and resolve them. For practical reasons, the first phase is done in parallel and the second phase is done sequentially or partially in parallel. This category of algorithms roughly work as explained in Algorithm 2.

Algorithm 2 Speculation and Conflict Resolution

```
1: procedure SPECCRPARALLELGRAPHCOLOURING(G(V,E))
2:     Initialization
3: speculation:
4:     for each v ∈ V do                                   ▷ Can be done in parallel
5:         colour[v] ← speculatedColour
6: conflict resolution:                                    ▷ Done serially or partially in parallel
7:     for each v ∈ V do
8:         if colour[v] has a conflict then                ▷ Conflicts can be found in parallel
9:             colour[v] ← a new colour which resolves the conflict    ▷ Greedy?
```

2.4 Algorithms

In this section, we will discuss some of the existing graph colouring algorithms belonging to both categories, Vertex Independent Set and Colouring and Speculation and Conflict Resolution, as discussed in the previous section. We also concern ourselves with only those parallel algorithms which are scalable. Hence algorithms like Parallel First Fit graph colouring, the parallel version of the First Fit colouring heuristic, are not considered.

2.4.1 Sequential Greedy Graph Colouring

We start with discussing a sequential graph colouring algorithm, one of the easiest, the Greedy Colouring algorithm. Many other algorithms, including many parallel graph colouring algorithms, are based on greedy colouring.

In Greedy Colouring, you choose each vertex of the graph and assign it the smallest colour number available which is not currently in use by one of its adjacent vertices. This is described in Algorithm 3. As is evident, the colour quality produced by Greed Colouring can be arbitrary. In other words, the colour quality produced will depend on the order in which vertices are processed by the algorithm.

Greedy Colouring can produce a colour quality of $\chi(G)$ for atleast one ordering of the vertices. But, on an average, this heuristic performs far from optimal. But the greedy colouring algorithm gives us an upper bound on the number of colours that it uses. The colouring produced uses at most $\Delta + 1$ colours, where Δ is the maximum

degree among all the vertices of the graph. AS the order in which the vertices are processed is very important, there have been many approaches suggested over the years which on an average produces a better colour quality. One of them is the so called Welsh-Powell Algorithm, in which we process the vertices in the order of their degrees.

Algorithm 3 Sequential Greedy Graph Colouring

```
1:  procedure GREEDYCOLOURING(G(V,E))
2:      Initialization
3:      for each v ∈ V do
4:          for each u ∈ V adjacent to v do
5:              if colour[u] ≠ 0 then
6:                  availableColours[colour[u]] ← FALSE
7:          for i from 1 to Δ + 1 do
8:              if availableColours[i] is TRUE then
9:                  colour[v] ← i
10:                 break
11:         Re-initialize availableColours[] array to TRUE
```

2.4.2 Parallel VIS Based Colouring

Here, we consider an algorithm belonging to category 1 algorithms. As discussed earlier, this involves finding Vertex Independent Sets and colouring those VIS in parallel. Now, we will introduce two more terms:

Maximal Vertex Independent Set: A Vertex Independent Set, γ, is said to be a Maximal Vertex Independent Set of a graph $G(V, E)$ if,

$$\forall \eta, \gamma \not\subset \eta$$

where η is a valid Vertex Independent Set of $G(V, E)$. It follows that there can be multiple Maximal Vertex Independent Sets.

Maximum Vertex Independent Set: A Vertex Independent Set, γ, is said to be a Maximum Vertex Independent Set of a graph $G(V, E)$, if γ is a Maximal Independent Set and,

$$|\gamma| = max_{\eta}|\eta|$$

where η is a Maximal Vertex Independent Set of $G(V, E)$. It follows that there can be multiple Maximum Vertex Independent Sets.

Ideally, we want to find a Maximum Vertex Independent Set of the graph in each iteration and colour those vertices in parallel. But finding Maximum Independent Sets of a graph is an NP-Complete problem. So, we have to instead go for non-optimal solutions. We consider the parallel algorithm suggested by Luby (1985) which finds Maximal Vertex Independent Sets of a graph in parallel.

In Luby's algorithm, every node is first assigned with some random number. Now, in each iteration, the random number assigned to each node is compared to its neighbours, done in parallel, to see if it is the local maximum, in which case, that node is added to a set *S*. At the end of each iteration, *S*, is a Maximal Vertex Independent Set of the graph *G(V,E)* and the vertices in *S* are removed from *V*. The set is *S* is emptied before a new iteration. This is a very simple algorithm to generate Maximal Vertex Independent Sets of a graph as depicted in Algorithm 4.

Algorithm 4 Maximal Vertex Independent Set

```
1:  procedure MAXIMALSET(G(V,E))
2:      Initialization                    ▷ randomNumber[] is initialized only once
3:      S ← φ
4:      for each v ∈ V do                                          ▷ Done in parallel
5:          S ← S ∪ v
6:          for each u ∈ V such that u is adjacent to v do
7:              if randomNumber[u] ≥ randomNumber[v] then
8:                  S ← S − v
9:                  break
10:     V ← V − S
11:     return S
```

Jones and Plassmann (1993) introduced a parallel graph colouring algorithm based on Luby's Maximal Vertex Independent Set algorithm. By their algorithm, in each iteration, we find a Maximal Vertex Independent Set of the graph using Luby's algorithm and then colour all the vertices in the set found, in parallel, using a single colour. Each iteration uses a different colour. We do this iteratively until all the vertices of the graph are coloured. This is explained in Algorithm 5

Algorithm 5 Jones-Plassmann-Luby Parallel Colouring Heuristic

```
1: procedure PARALLELCOLOURING(G(V,E))
2:     Initialization
3:     n ← 0
4:     currentColour ← 1
5:     graphSize ← |V|
6:     while n ≠ graphSize do
7:         S ←MaximalSet(V,E)
8:         for each v ∈ S do                          ▷ Done in parallel
9:             colour[v] ← currentColour
10:        currentColour ← currentColour + 1
11:        n ← n + |S|
```

2.4.3 Parallel Conflict Resolution Based Colouring

Here, we consider two algorithms belonging to category 2 algorithms. It involves two phases, the first colouring phase with potential conflicts and the second phase where these conflicts are resolved either sequentially or partially in parallel.

2.4.3.1 Speculation and Colouring

Here, we discuss an algorithm presented by Gebremedhin and Manne (2000) which instead of finding Maximal Vertex Independent Sets in each iteration, relaxes the condition, so that we find and colour sets which are not really independent sets in each iteration possibly incurring conflicts. These conflicts are then identified in parallel in phase 2. In phase 3, we re-colour the vertices identified with conflicts sequentially.

This involves an initial graph partitioning phase, during which we partition the graph into *n* parts, where *n* is the number of processors/cores we have. Then each processor/core takes up each partition and then colour them using some sequential colouring method. At the end of this phase 1, we thus have a colouring with possible conflicts at the partition boundaries. In phase 2, we identify these conflicts in parallel. In phase 3, we re-colour these vertices with conflicts sequentially. The scheme is presented as in Algorithm 6.

We also have a GPU based graph partitioning, speculation and conflict resolution algorithm by Grosset *et al.* (2011). The graph is first partitioned in the CPU. The parti-

Algorithm 6 Partitioning, Speculation and Conflict Resolution

```
1: procedure PARTITIONCOLOURING(G(V,E))
2:     Initialization
3:     G(V,E) is partitioned into n partitions V_1 to V_n        ▷ Each of n threads gets one
4: Phase 1 (Partition Colouring):
5:     for each v ∈ V_i do                                       ▷ Done in parallel by n threads
6:         colour[v] ← A colour by some sequential colouring algorithm
7: Phase 2 (Conflict Detection):
8:     conflictSet ← φ
9:     for each v ∈ V do                                         ▷ Done in parallel
10:        for each u ∈ V such that u and v are adjacent do
11:            if colour[v] = colour[u] then
12:                conflictSet ← conflictSet ∪ min(v, u)
13: Phase 3 (Conflict Resolution):
14:    for each v ∈ conflictSet do
15:        colour[v] ← A colour by some sequential colouring algorithm
```

tions are then coloured using some sequential colouring heuristics on the GPU. At the end of this phase, the potential conflicts are found in parallel at the boundary vertices. In the next iteration, these conflicts are recoloured possibly generating other conflicts. This process is continued until the total number of conflicts are below some threshold. And then the rest of the conflicts are resolved sequentially. This is depicted in Algorithm 7.

2.5 Our Approach

In our work, we try a number of options like `RANDCOLOURING` and `MINMAXCOLOURING` which uses some of the suggestions by Cohen and Castonguay (2012) on top of the algorithm put forward by Jones and Plassmann (1993). Like every other implementation, the data structures used are really important. Especially in our case, as we are using a GPU, the structure and size of data copied to and stored on the GPU are very important. From here on, n represents the number of vertices and m represents the number of edges in the graph.

Algorithm 7 GPU: Partitioning, Speculation and Conflict Resolution

1: **procedure** PARTITIONCOLOURING(G(V,E))
2: *Initialization*
3: *CPU*:
4: *G(V,E) is partitioned into n partitions* V_1 *to* V_n ▷ Each of n threads gets one
5: *GPU (Partition Colouring)*:
6: **for** $each\ v \in V_i$ **do** ▷ Done in parallel by n threads
7: $colour[v] \leftarrow$ *A colour by some sequential colouring algorithm*
8: *GPU (Conflict Detection)*:
9: *conflictSet* $\leftarrow \phi$
10: **for** $each\ v \in V$ **do** ▷ Done in parallel
11: **for** $each\ u \in V$ such that u and v are adjacent **do**
12: **if** $colour[v] = colour[u]$ **then**
13: *conflictSet* $\leftarrow$ *conflictSet* $\cup\ min(v, u)$
14: **if** $|conflictSet| < threshold$ **then**
15: *goto CPU (Conflict Resolution)*
16: *GPU (Conflict Resolution)*:
17: **for** $each\ v \in$ *conflictSet* **do** ▷ Done in parallel
18: $colour[v] \leftarrow$ *A colour by some sequential colouring algorithm*
19: *goto GPU (Conflict Detection)*
20: *CPU (Conflict Resolution)*:
21: **for** $each\ v \in$ *conflictSet* **do**
22: $colour[v] \leftarrow$ *A colour by some sequential colouring algorithm*

2.5.1 CSR: Compressed Sparse Row Representation

Graphs are usually stored in an adjacent matrix representation or an adjacency list representation. Both have their own pros and cons. In our case, as we have to copy the entire graph, which are itself huge, to the GPU, we can't have the luxury of adjacent matrix representation as it requires $O(n^2)$ storage space. As we are dealing with GPUs, which don't really provide enough support to use connected lists, adjacency list representation ($O(m+n)$) in its classical form is also not practical. So, we use an array based adjacency list representation called Compressed Sparse Row Representation.

> Compressed Sparse Raw Representation of a graph involves the use of just three arrays (two in case the edges are of unit weight). The first array, called the Offsets Array, has a size of the number of vertices. The second array, called the Edges Array, and the third array, called the Weights Array, have the size of the number of edges each.

With respect to our problem, the third array, Weights Array, is not used as all the edges are assumed to be of unit weight. So, CSR takes up $O(m+n)$ space, but uses only arrays which are easier to work with on GPUs.

Offsets Array

The offsets array (`offsetArray[]`) has *n* (no. of vertices) elements. Each of these elements represent the offset in the Edges Array where the edges adjacent to the respective vertices are stored. So, `offsetArray[i]` represents the offset in the `edgesArray[]` where the edges adjacent to vertex $i+1$ (assuming vertices are number 1 to n) are stored from. Therefore, it follows that the adjacent edges of the vertex $i+1$ will be stored from index `offsetArray[i]` upto, but not including, `offsetArray[i+1]` for all $i < n-1$ and for $i = n-1$, from index `offsetArray[i]` upto, but not including, `m`. It also follows that if a vertex $i+1$ is of degree 0, that is if a vertex doesn't have any adjacent vertices, then

`offsetArray[i]` = `offsetArray[i+1]` (`m` if $i = n-1$).

Edges Array

The edges array (`edgesArray[]`) has m (no. of edges) elements. All these m elements represent one of the n vertices. For a vertex, $i + 1$, the `edgesArray[]` stores its adjacent vertices from index `offsetArray[i]` upto, but not including, `offsetArray[i+1]` for all $i < n-1$ and for $i = n-1$, from index `offsetArray[i]` upto, but not including, `m`.

Undirected Graphs in CSR: UCSR

It is to be noted that CSR is predominantly used for directed graphs. It can be used as such for undirected graphs too, but that might cause some extra processing to be done to find the neighbourhood of a vertex in the graph and might lead to a data race in parallel computing. So, one way around this is to include all edges of a graph as directed edges in both directions. That is, given an undirected graph $G(V, E)$ with m edges, our `edgesArray[]` graph will have $2m$ elements instead of m elements. But the order of space remains the same. We call this *Undirected CSR* or *UCSR*.

Example

Here, we take an example to explain how CSR works. We take both a directed graph and an undirected graph to show how we use the CSR.

Consider the directed graph with $n = 4$ and $m = 4$ as given in the table 2.1. It's corresponding CSR is given in the figure 2.1.

From Node	To Node
1	2
1	3
2	3
2	4

Table 2.1: A Directed Graph with $n = 4$ and $m = 4$

Consider the undirected graph with $n = 4$ and $m = 3$ as given in the table 2.2. It's corresponding CSR is given in the figure 2.2 and it's UCSR is given in the figure 2.3.

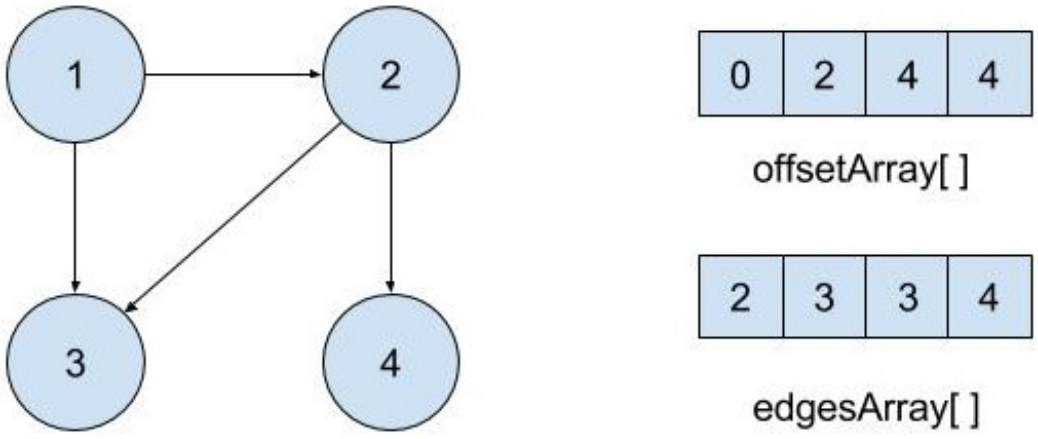

Figure 2.1: The Directed Graph from 2.1 and its CSR

Note that the `edgesArray[]` has $2m = 6$ elements in the UCSR.

Incident Node	Incident Node
1	2
1	4
2	3

Table 2.2: An Undirected Graph with $n = 4$ and $m = 3$

2.5.2 RANDCOLOUR: Random Colouring And Conflict Resolution

We started with a random colouring and conflict resolution based algorithm which we call `RANDCOLOUR`. This is an algorithm pertaining to category 2 algorithms. We first speculate the colours randomly and then find the conflicts on the GPU, both of which are done in parallel. And then we resolve the conflicts sequentially on the CPU. As we know a graph $G(V, E)$ with maximum degree Δ can be coloured with atmost $\Delta + 1$ colours, RANDCOLOUR is an attempt at a fast $\Delta + 1$ colouring using GPU. `RANDCOLOUR` algorithm almost always produces a colour quality of $\Delta + 1$ and was aimed at a fast colouring compromising colour quality in the process.

The scheme is given in Algorithm 8. First we find Δ, the maximum degree of the graph. This task can be done in parallel on the GPU. With UCSR of the graph, the

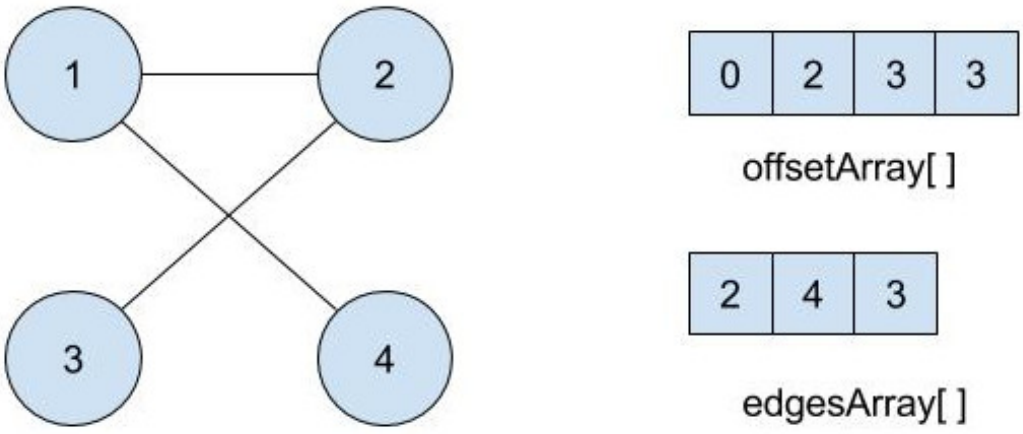

Figure 2.2: The Undirected Graph from 2.2 and its CSR

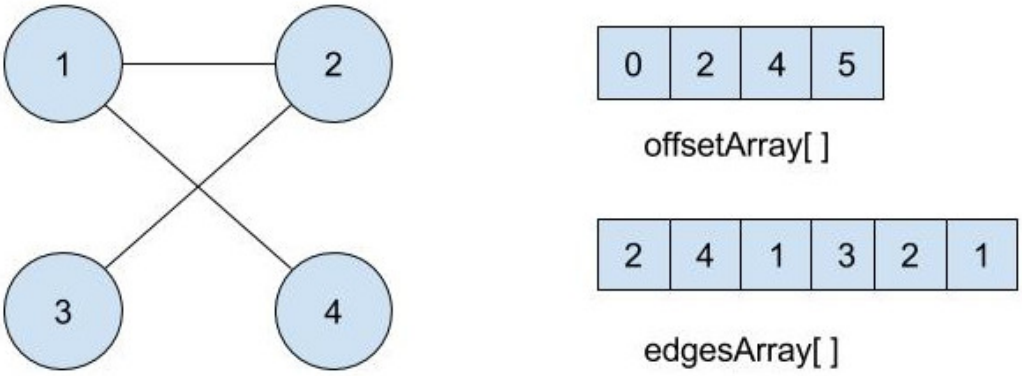

Figure 2.3: The Undirected Graph from 2.2 and its UCSR

degree of a vertex $i+1$ is given by `offsetArray[i+1]-offsetArray[i]` ($m-$ `offsetArray[i]` if $i = n-1$). Once we have Δ, we use `CURAND` library, which generates random numbers really fast using CUDA capable GPU, to colour the graph with random integers ranging from 1 to $\Delta + 1$. Once this colouring is over, we try to find out the conflicts that were inflicted. This can be done in parallel on the GPU too. We mark all the conflicts and then resolve them on the CPU using a greedy approach.

Algorithm 8 RANDCOLOUR

procedure RANDOMCOLOURING(G(V,E))
    Initialization
GPU:
    Initialize MaxDegree ▷ Found in parallel on GPU
    for $each\ v \in V$ **do** ▷ Done in parallel by |V| threads
        $colour[v] \leftarrow CURAND(MaxDegree + 1) + 1$ ▷ RAND on GPU
    conflictSet $\leftarrow \phi$
    for $each\ v \in V$ **do** ▷ Done in parallel
        for $each\ u \in V$ such that u and v are adjacent **do**
            if $colour[v] = colour[u]$ **then**
                conflictSet $\leftarrow$ *conflictSet* $\cup\ min(v, u)$
CPU:
    for $each\ v \in$ *conflictSet* **do**
        $colour[v] \leftarrow$ *A colour by some greedy colouring algorithm*

2.5.3 MINMAXCOLOUR: Maximal VIS And Colouring

We then tried a version of the algorithm put forward by Jones and Plassmann (1993) which uses some of the suggestions by Cohen and Castonguay (2012). As discussed earlier, the algorithm by Jones and Plassmann (1993) is based on our 1 set of algorithms. So, here, we try to find Maximal Vertex Independent Sets and then colour them in parallel.

Improvements

- The original version uses a `Max` approach for finding a Maximal Vertex Independent Set. A `Min` approach also gives us a Maximal Vertex Independent Set which is mutually exclusive with the `Max` based set (except when a vertex doesn't have any adjacent vertices; discussed in the next point). So, we find two Maximal Vertex Independent Sets in each iteration, one based on `Max` approach and the other on `Min` approach. Then, we colour both sets with two different colours in each iteration. This way we get to speed up the colouring by reducing the number of iterations required.
- Conflicts can arise when two neighbouring vertices have the same random number allotted to them. That same random number could be a neighbourhood minimum or maximum in which case we have multiple candidates for the `Max` set or `Min` set from the same neighbourhood. If a vertex doesn't have any neighbours, it can possibly be added to the `Max` set as well as `Min` set. To avoid conflicts between the `Max` based set and `Min` based set in each iteration, we follow a set of conventions as follows:
  - In each iteration, the `Max` based set takes the smaller of the two colour numbers allowed in that iteration.

- If two neighbours have the same random number allotted to them, the vertex with a smaller index has higher precedence for `Max` set.
- If two neighbours have the same random number allotted to them, the vertex with a higher index has higher precedence for `Min` set.
- If a vertex doesn't have any neighbours, it is considered a local maxima and is added to `Max` set. This is because `Max` sets get a lower colour number compared to `Min` sets in an iteration as discussed earlier.

The scheme for this is given in Algorithms 9 and 10. In Algorithm 9, we have the Min-Max Maximal Independent Set algorithm which is an improved version of Luby (1985). The two inner for loops in the algorithm at line 7 and line 17 can be merged together using some conditionals. We find two maximal sets in each iteration. In the colouring part, in each iteration, we colour the Max set with a colour c and the Min set with $c + 1$. The algorithm is iterated until there are no vertices left to colour. The maximal sets are found in parallel and they can be coloured in the same kernel call.

The Maximal VIS based algorithm might be a little more time consuming as it employs an iterative approach, but should produce better colour quality. The main reason why this approach was our choice is because the entire process happens in parallel here, that is there is no sequential colouring component. As we use GPU, we try to maximize the use of the massive parallelism GPUs offer.

Algorithm 9 Min-Max Maximal Vertex Independent Sets

```
1: procedure MinMaxMaximalSet(G(V,E))
2:     Initialization                          ▷ randomNumber[] is initialized only once
3:     MinS ← φ
4:     MaxS ← φ
5:     for each v ∈ V do                                          ▷ Done in parallel
6:         MaxS ← MaxS ∪ v
7:         for each u ∈ V such that u is adjacent to v do
8:             if randomNumber[u] > randomNumber[v] then
9:                 S ← S − v
10:                break
11:            else
12:                if randomNumber[u] = randomNumber[v] then
13:                    if u > v then
14:                        S ← S − v
15:                        break
16:        MinS ← MinS ∪ v
17:        for each u ∈ V such that u is adjacent to v do
18:            if randomNumber[u] < randomNumber[v] then
19:                S ← S − v
20:                break
21:            else
22:                if randomNumber[u] = randomNumber[v] then
23:                    if u < v then
24:                        S ← S − v
25:                        break
26:    MinS ← MinS − MaxS
27:    V ← V − MaxS
28:    V ← V − MinS
29:    return MaxS, MinS
```

Algorithm 10 MINMAXCOLOUR

```
1: procedure MinMaxColouring(G(V,E))
2:     Initialization
3:     n ← 0
4:     currentColour ← 1
5:     graphSize ← |V|
6:     while n ≠ graphSize do
7:         MaxS, MinS ←MinMaxMaximalSet(V,E)
8:         for each v ∈ MaxS do                                   ▷ Done in parallel
9:             colour[v] ← currentColour
10:        for each v ∈ MinS do                                   ▷ Done in parallel
11:            colour[v] ← currentColour + 1
12:        currentColour ← currentColour + 2
13:        n ← n + |MaxS| + |MinS|
```

CHAPTER 3

PARALLEL GRAPH COLOURING: INCREMENTAL

3.1 Why Incremental?

Graphs are being used in a varied lot of applications these days. Graphs are ever more important and ever growing. Only a very small share of practical graphs are static. Most of them keep on changing. Edges and Vertices get added and deleted every now and then. It is not a great idea to run a computationally intensive algorithm again on a graph just because a few thousands (a small fraction compared to total graph size) of edges are added to it. Thus incremental approaches are very important so as to save on computation and time.

3.2 Handling a Growing Graph

So, with incremental graph colouring, we are accommodating additions of vertices and edges. We assume without any loss of functionality that no new vertices are added. Only edges are added. This is easy to see as adding a new vertex (with some edges incident on it) is equivalent to adding edges to a vertex with zero degree. As long as we have an idea about the upper bound on the number of vertices through the applications of the graph, we should be fine.

Through the additions of edges, the `offsetArray[]` of UCSR doesn't grow in size. But the size of `edgesArray[]` will grow. So, we should have a reasonable upper bound on the number of edges that can be incident on each vertex. We can set the extra `edgesArray[]` elements to zero initially. When an edge is added, the new edge's information can be added to the `edgesArray[]` at these elements which are set to zero.

CHAPTER 4

PARALLEL GRAPH COLOURING: DECREMENTAL

4.1 Why Decremental?

As discussed in the last chapter, almost all practical graphs keep on changing. Edges and Vertices get added and deleted often. Deletion of a small fraction of edges compared to total graph size shouldn't call for running a computationally intensive algorithm all over again again on the graph. Decremental approach helps save running time and avoids unnecessary computations.

4.2 Handling a Shrinking Graph

With decremental graph colouring, we are removing vertices and edges. Without any loss of functionality, we can choose not to remove vertices. Only edges are deleted. Removing a vertex (with some edges incident on it) is equivalent to removing all the edges incident on that vertex.

So, through the deletions of edges, the `offsetArray[]` of UCSR doesn't change in size as we don't really remove vertices. But the size of `edgesArray[]` can change. As we want the size of `edgesArray[]` to not change, when an edge is deleted, the new edge's information can be removed from the `edgesArray[]` by setting those elements to zero.

CHAPTER 5

EXPERIMENTAL EVALUATION

5.1 Experimental Setup

5.2 Test Data

5.3 Parallel Graph Colouring on GPU

5.4 Incremental Parallel Graph Colouring on GPU

5.5 Decremental Parallel Graph Colouring on GPU

CHAPTER 6

CONCLUSION AND FUTURE WORK

REFERENCES

1. **ÇAtalyüRek, ı. V.**, **J. Feo**, **A. H. Gebremedhin**, **M. Halappanavar**, and **A. Pothen** (2012). Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Comput.*, **38**(10-11), 576–594. ISSN 0167-8191. URL http://dx.doi.org/10.1016/j.parco.2012.07.001.

2. **Cohen, J.** and **P. Castonguay** (2012). Efficient graph matching and coloring on the gpu. *GTC on-demand S0332*.

3. **Gebremedhin, A. H.** and **F. Manne** (2000). Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, **12**(12), 1131–1146. ISSN 1096-9128. URL http://dx.doi.org/10.1002/1096-9128(200010)12:12<1131::AID-CPE528>3.0.CO;2-2.

4. **Grosset, A. V. P.**, **P. Zhu**, **S. Liu**, **S. Venkatasubramanian**, and **M. Hall** (2011). Evaluating graph coloring on gpus. *SIGPLAN Not.*, **46**(8), 297–298. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2038037.1941597.

5. **Jones, M. T.** and **P. E. Plassmann** (1993). A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, **14**(3), 654–669. ISSN 1064-8275. URL http://dx.doi.org/10.1137/0914041.

6. **Luby, M.**, A simple parallel algorithm for the maximal independent set problem. *In Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85. ACM, New York, NY, USA, 1985. ISBN 0-89791-151-2. URL http://doi.acm.org/10.1145/22145.22146.

7. **Naumov, M.**, **P. Castonguay**, and **J. Cohen** (2015). Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. *NVIDIA Technical Report*, **001**. URL https://research.nvidia.com/publication/parallel-graph-coloring-applications-incomplete-lu-factorization-gpu.

8. **NVIDIA Corporation** (2016). Cuda toolkit documentation. URL http://docs.nvidia.com/cuda/parallel-thread-execution/#ptx-machine-model. [Online; accessed on 22 April, 2016].

9. **Sengupta, S.**, Parallel graph coloring algorithms on the gpu using opencl. *In Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*. 2014.

10. **Sutter, H.** (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, **30**(3). URL http://www.gotw.ca/publications/concurrency-ddj.htm.

11. **Wikipedia, the free encyclopedia** (2016). Moore's law. URL https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg. [Online; accessed on 22 April, 2016].