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 `Shamil` what he is today.

Thank God.

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.

To be filled.

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

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 supercomputers 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.

CHAPTER 2

Background

2.1 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.

2.2 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 (Jensen and Toft, 2011). In other words, it's 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.

2.2.1 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 its 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. An example of Vertex Colouring is given in Figure 2.1. Vertex *v4* is coloured initially with blue (*b)* colour. Then, its neighbours, vertices *v3, v5, v6 and v2* are coloured with red (*r)* colour. As *v1* has *v2 and v3* also as its edges, it can't take colour red (*r*). So, it's coloured with a new colour, green (*g*).

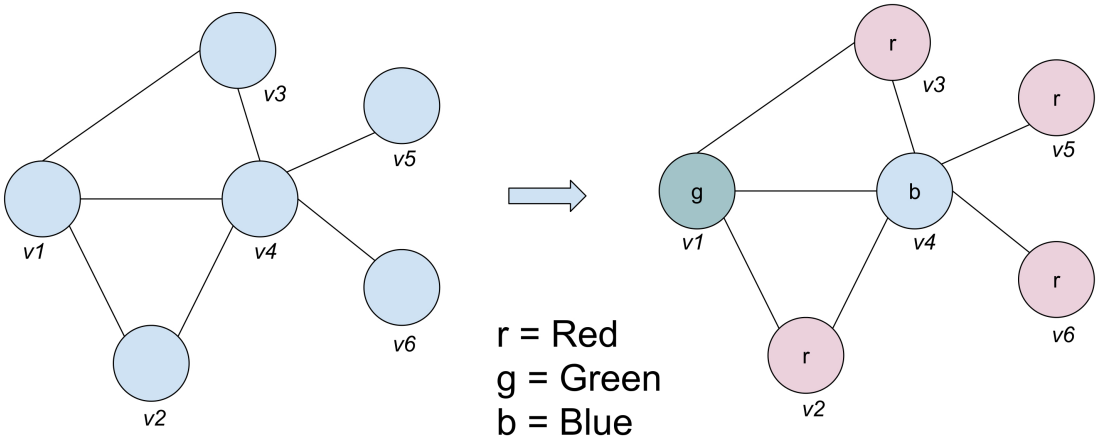

Figure 2.1: Vertex Colouring of a given graph using 3 colours.

2.2.2 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* (Appel and Haken, 1977; Thomas, 1998; Wilson, 2014), 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.

For example, $\chi(G)$ of the graph given in Figure 2.1 is 3. Also, it follows that if a graph is *k*-colourable, we have:

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

2.2.3 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.

2.2.4 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).

2.2.5 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 (Leighton, 1979)
- Register allocation problem during compilation (Briggs *et al.*, 2004)
- Solving Sudoku (Herzberg and Murty, 2007)
- Applications on biological networks like Protein-Protein Interaction (PPI) Networks (Khor, 2009)
- Scheduling cell-transmissions in an ATM switch (Lakshman *et al.*, 1994)

2.3 Parallelization

2.3.1 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 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 clocks or higher clock speeds, multi-core processors started rolling out.

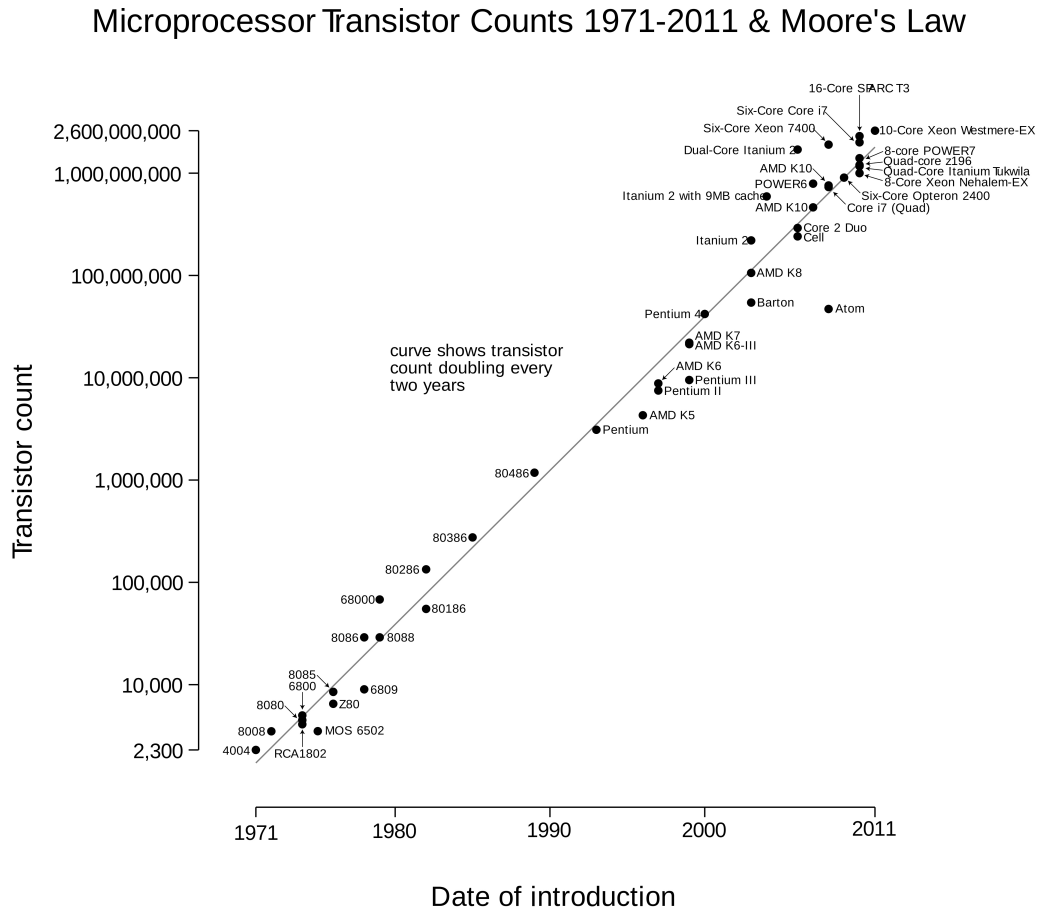

Figure 2.2: Graph showing Moore's Law in action, from Wikipedia, the free encyclopedia (2016). Each data point is a processor.

2.3.2 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, 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 2.3.

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 $1/n^{th}$

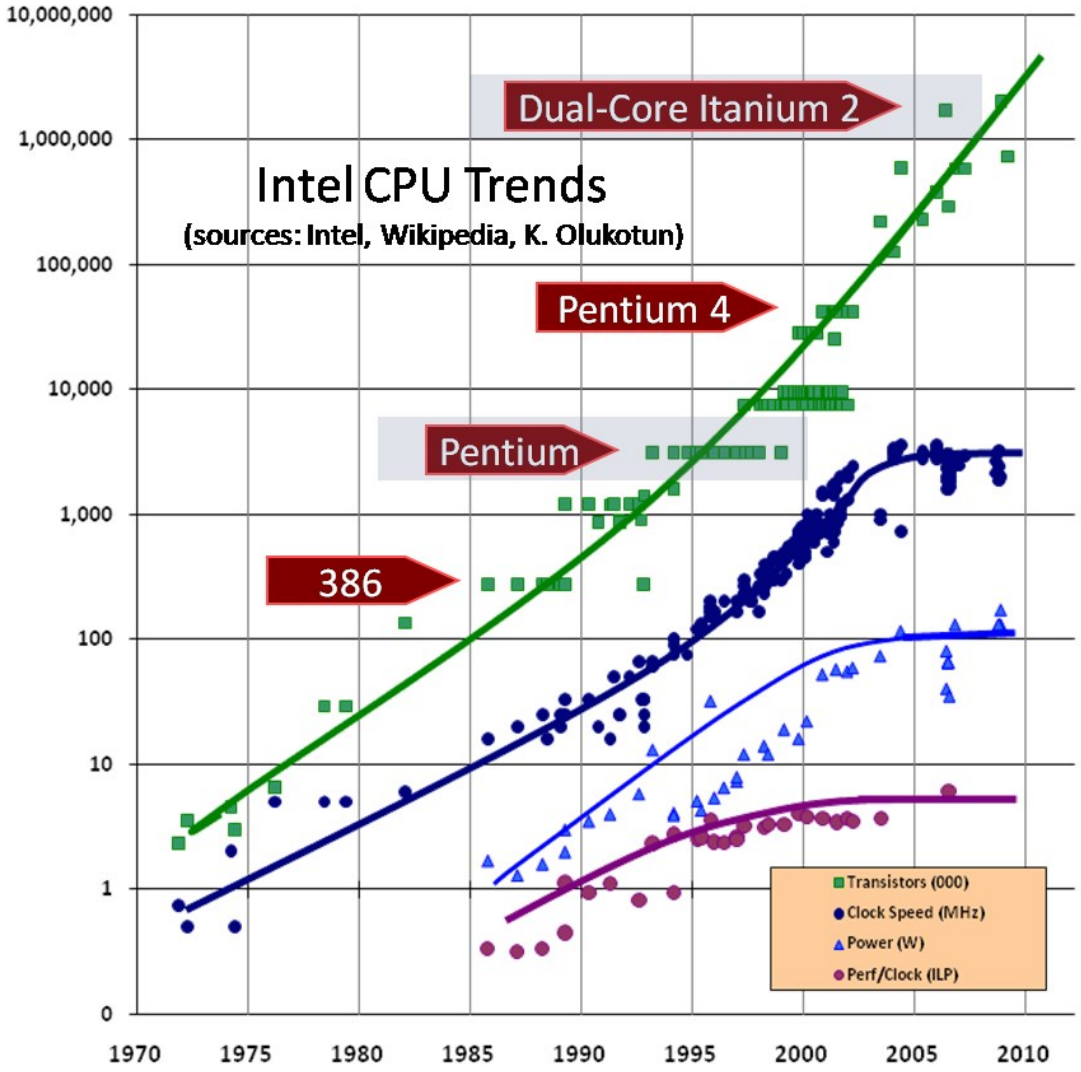

Figure 2.3: *Intel*'s transition from single core processors to multi-core processors around 2004-2005, from Sutter (2005).

compared to the running time when run on a single core processor. Though we rarely really reach this ideal speedup, as stated by Amdahl's law (Amdahl, 1967) etc., we still achieve significant speed ups.

2.3.3 Parallelization 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 algo-

rithms 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 $\epsilon > 0$ (Zuckerman, 2006). 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 (Bozdag *et al.*, 2008, 2005) and supercomputers (ÇAtalyüRek *et al.*, 2012). 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.

2.4 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 supercomputers had a very high number of cores and they came at a price.

2.4.1 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 supercomputers. They also came with thousands of cores. But they were specialized for graphics related operations. Then came the paradigm of GPGPU, General Purpose computing on Graphics Processing Units. 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.

2.4.2 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

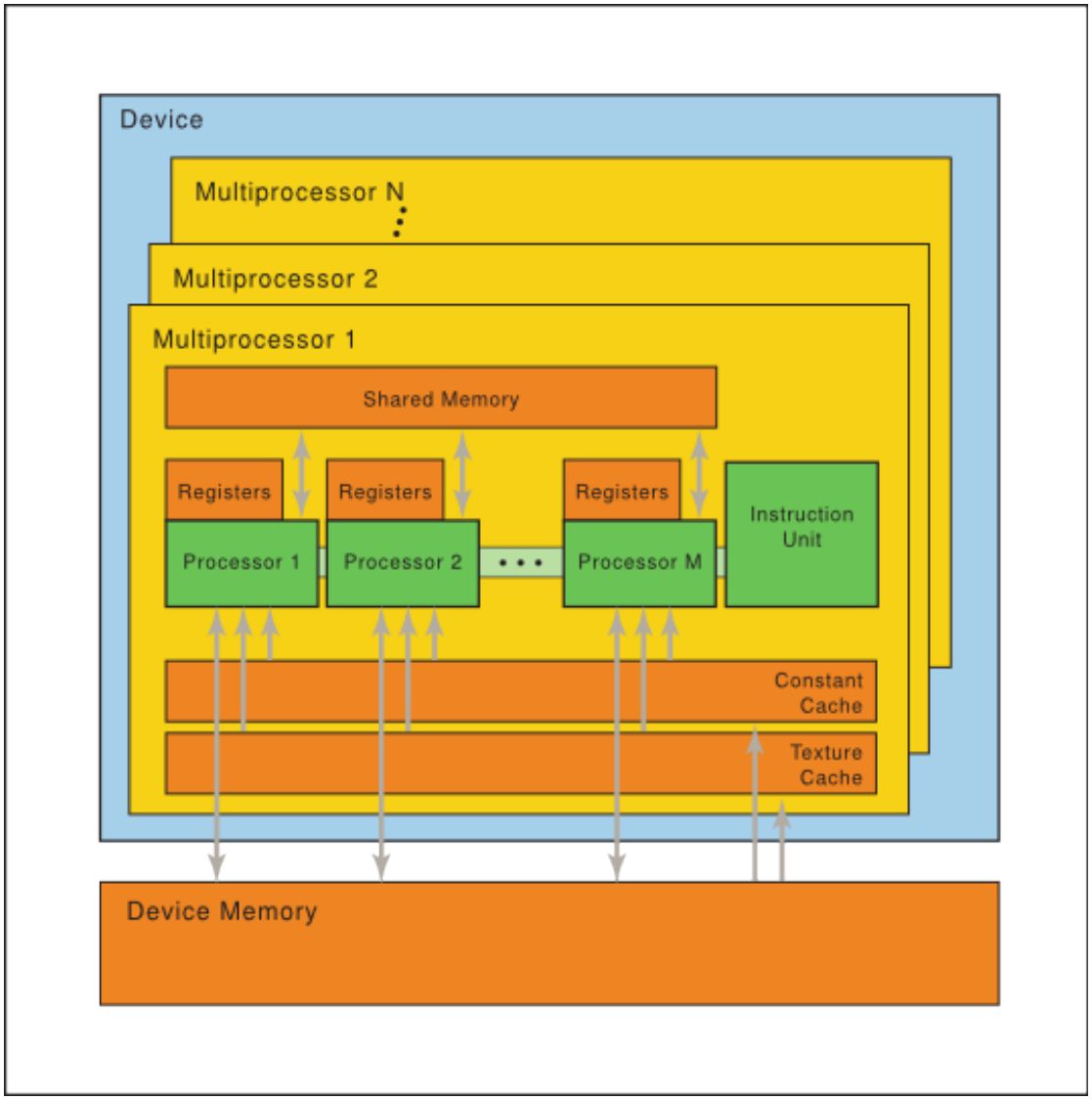

Figure 2.4: NVIDIA GPU hardware model, from NVIDIA Corporation (2016*a*).

In an NVIDIA GPU, as shown in the figure 2.4, 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.

2.4.3 Challenges

Though, GPUs let us access thousands of threads easily, it comes with a cost (Lee *et al.*, 2010). In discrete GPUs like the NVIDIA GPUs used in our work, 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 Bus, which is not super fast with bit rates at around `5 Gbits/sec`, and the fact that while transferring data, we force the CPU to wait for the GPU, 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 (Hovland, 2008). So, as a GPU programmer, one must try to reduce data transfer between the CPU and the GPU. Also, GPUs are not yet ready to do everything that a normal CPU does.

2.5 Incremental/Decremental

As we discussed already, Graph Colouring is a very important problem 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 is 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 addition/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 3

PARALLEL GRAPH COLOURING

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

3.1 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.*

3.2 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 Bozdag *et al.* (2008, 2005) and ÇAtalyüRek *et al.* (2012) are based on distributed computing setups, supercomputers 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.

3.3 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:

3.3.1 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 straightforward. 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 works as explained in Algorithm 1.

Phase 1 of the Algorithm 1 is done in Line 5 where we find a VIS of the graph. This step can be parallelized as discussed later in Algorithm 4 and Algorithm 9. Phase 2 of the algorithm, colouring the newly found VIS of the graph, is done in Lines 6 and 7. In Line 8, we remove the vertices which were coloured in the current iteration from the graph. We then choose a new colour for the next iteration as in Line 9 and return to Line 4. The algorithm iterates until there are no more vertices to be processed.

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
```

3.3.2 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 could colour the vertices in the VIS found in each step without any conflict. This second category of algorithms instead lets 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 works as explained in Algorithm 2.

Phase 1 of the Algorithm 2, *speculation*, is done in Lines 4 and 5. We are not explicitly mentioning how this speculation is done as there are many possible ways to speculate the colours. Some of the ways to speculate colours are discussed in Algorithms 6, 7 and 8. Conflicts are then resolved, mostly using some form of Greedy Colouring, in the phase 2 of the algorithm as in Lines 7, 8 and 9. Line 8 assumes conflicts are already detected. This *Conflicts Detection* part can be parallelized.

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
```

3.4 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.

3.4.1 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 Greedy Colouring can be arbitrary. In other words, the colour quality produced will depend on the order in which vertices are processed by the algorithm.

Algorithm 3 proceeds by choosing each vertex in some order, Line 3, and colouring them in a greedy fashion. The order in which the vertices are chosen is very important in deciding the colour quality. There are many ways to decide the order in which the vertices are processed and we assume one of the ways is chosen. We then check the vertex's neighbourhood, Line 4, and if a neighbour is already coloured, Line 5, we mark its colour as unavailable as in Line 6. Now, for the vertex, we check which is the smallest colour less than or equal to $\Delta + 1$ (Greedy Colouring uses at most $\Delta + 1$ colours, where Δ is the maximum degree among all the vertices of the graph) that is available. This is done in Lines 7 and 8. Then finally we colour the vertex with this new colour as in Line 9. The `availableColours[]` array is reinitialized to all *TRUE* values in Line 11 before the algorithm goes back to Line 3.

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 produce a better colour quality. One of them is the so called Welsh-Powell Algorithm (Welsh and Powell, 1967), 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
```

3.4.2 Parallel VIS Based Colouring

Here, we consider an algorithm belonging to the category covered by Algorithm 1. 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.

We assume random numbers are generated and stored in `randomNumber[]` as in Line 2. We consider all the vertices of the graph in parallel as in Line 4. In Lines 6 to 9, we check if the current vertex is indeed a local maxima and if so, it is added to the Maximal Set *S* which is then returned after all threads are synchronized.

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:     Synchronize
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.

In each iteration until all the vertices are processed (Line 6), we find a Maximal VIS in Line 7 and colour them as in Lines 8 and 9. We then choose a new colour for the next

iteration as in Line 10, remove the vertices that were coloured in the current iteration as in Line 12 and return to Line 6. The algorithm iterates until there are no more vertices to be processed.

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                          ▷ Or V ≠ φ
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|
12:         V ← V \ S
```

3.4.3 Parallel Conflict Resolution Based Colouring

Here, we consider two algorithms belonging to the category covered by Algorithm 2. 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.

3.4.3.1 Conflict Resolution: Sequential

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. Each processor/core takes up each partition and then colours 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 in Algorithm 6.

In Line 3, the graph $G(V, E)$ is partitioned into n partitions V_1 *to* V_n. In Lines 5 and 6, each thread colours its partition using some sequential colouring algorithm. In Lines 8 to 12, the set of vertices with colouring conflicts, `conflictSet`, is found. In Lines 14 and 15, the conflicts are resolved by recolouring the vertices in `conflictSet` using some sequential colouring algorithm.

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
```

3.4.3.2 Conflict Resolution: Partially in Parallel

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 partitions 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 in parallel possibly generating other conflicts. This process is continued until the total number of conflicts is below some threshold. Then the rest of the conflicts are resolved sequentially. This is depicted in Algorithm 7.

In Line 4, the graph $G(V, E)$ is partitioned into n partitions V_1 *to* V_n in the CPU. In Lines 6 and 7, each thread colours its partition using some sequential colouring

algorithm in the GPU. In Lines 9 to 13, the set of vertices with colouring conflicts, `conflictSet`, is found in the GPU. We then check if the number of conflicts is below some threshold in Line 14 in which case we proceed straight to the *Conflict Resolution* phase which happens in the CPU. If the number of conflicts is not below the threshold, we try to do a parallel conflict resolution in the GPU as in Lines 17 and 18 and then go back to the GPU *Conflict Detection* phase at Line 8. In Lines 21 and 22, the conflicts are resolved by recolouring the vertices in `conflictSet` using some sequential colouring algorithm in the CPU.

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 ∈ V_i do                                     ▷ Done in parallel by n threads
7:          colour[v] ← A colour by some sequential colouring algorithm
8:  GPU (Conflict Detection):
9:      conflictSet ← φ
10:     for each v ∈ V do                                       ▷ Done in parallel
11:         for each u ∈ V such that u and v are adjacent do
12:             if colour[v] = colour[u] then
13:                 conflictSet ← conflictSet ∪ {min(v, u)}
14:     if |conflictSet| < threshold then
15:         goto CPU (Conflict Resolution)
16: GPU (Conflict Resolution):
17:     for each v ∈ conflictSet do                             ▷ Done in parallel
18:         colour[v] ← A colour by some sequential colouring algorithm
19:     goto GPU (Conflict Detection)
20: CPU (Conflict Resolution):
21:     for each v ∈ conflictSet do
22:         colour[v] ← A colour by some sequential colouring algorithm
```

3.5 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.

3.5.1 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. Graphs in practical applications are usually huge. In our case, as we have to copy the entire graph to the GPU, we can't have the luxury of adjacent matrix representation as it requires $O(n^2)$ storage space. Also, we deal with a lot of sparse graphs and for sparse graphs, adjacent matrix representation is very wasteful in terms of space. For sparse graphs, adjacency list based representations are better. As we are dealing with GPUs, which don't really provide enough support for using 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 Offset Array, has a size of $n + 1$, where n is 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 $n + 1$ (n is the no. of vertices) elements. Each of these elements represents 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 $i < n - 1$. To extend this to vertex n, that is, $i = n - 1$, we initialize `offsetArray[n]` with the value `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]`

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 $i < n$.

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 3.1. Its corresponding CSR is given in the figure 3.1.

Consider the undirected graph with $n = 4$ and $m = 3$ as given in the table 3.2. Its

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

Table 3.1: A Directed Graph with $n = 4$ and $m = 4$

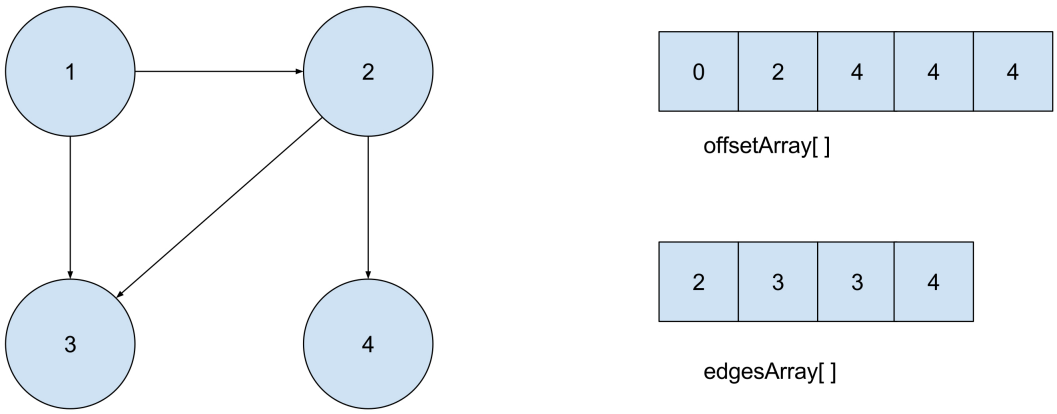

Figure 3.1: The Directed Graph from 3.1 and its CSR

corresponding CSR is given in the figure 3.2 and its UCSR is given in the figure 3.3. Note that the `edgesArray[]` has $2m = 6$ elements in the UCSR.

Incident Node	Incident Node
1	2
1	4
2	3

Table 3.2: An Undirected Graph with $n = 4$ and $m = 3$

3.5.2 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 the category covered by Algorithm 2. We first speculate the colours randomly and then find the conflicts on the GPU, both of which are done in parallel. 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

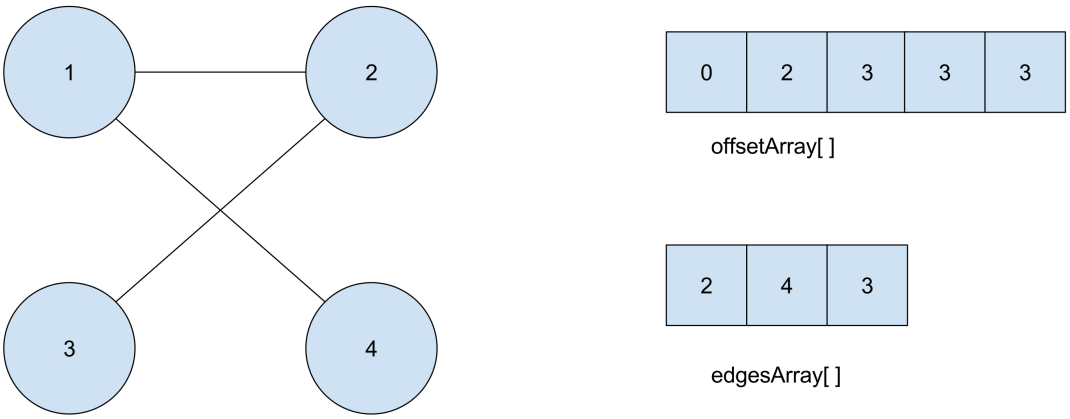

Figure 3.2: The Undirected Graph from 3.2 and its CSR

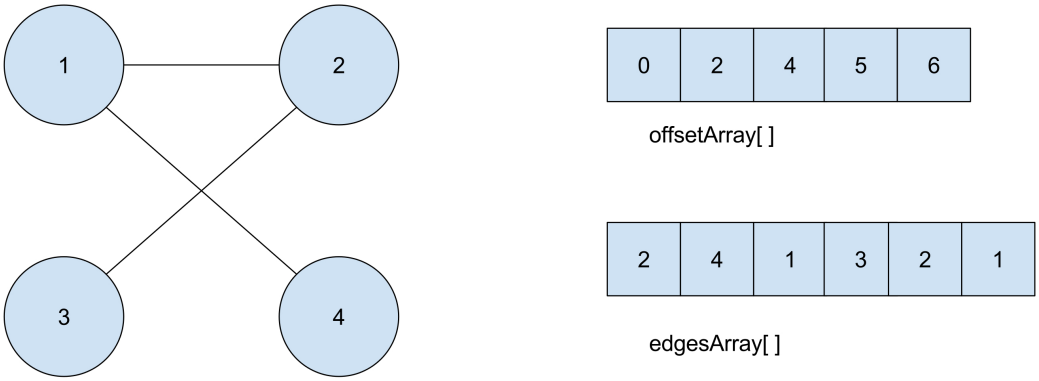

Figure 3.3: The Undirected Graph from 3.2 and its UCSR

atmost $\Delta + 1$ colours, RANDCOLOUR is an attempt at a fast $\Delta + 1$ colouring using GPU. `RANDCOLOUR` algorithm produces a worst 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 (Line 4). This task can be done in parallel on the GPU. With UCSR of the graph, the degree of a vertex $i + 1$ is given by `offsetArray[i+1]−offsetArray[i]`. Once we have Δ, we use `cuRAND` library (NVIDIA Corporation, 2016*b*), which generates random numbers really fast using CUDA capable GPUs, to colour the graph with random integers ranging from 1 to $\Delta + 1$ (Lines 5 and 6). 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 (Lines 8 to 12). We mark all the conflicts and then resolve them on the CPU

Algorithm 8 RANDCOLOUR

```
1: procedure RANDOMCOLOURING(G(V,E))
2:     Initialization
3: GPU (Speculation):
4:     Initialize MaxDegree                          ▷ Found in parallel on GPU
5:     for each v ∈ V do                             ▷ Done in parallel by |V| threads
6:         colour[v] ← CURAND(MaxDegree + 1) + 1     ▷ RAND on GPU
7: GPU (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: CPU (Conflict Resolution):
14:    for each v ∈ conflictSet do
15:        colour[v] ← A colour by some greedy colouring algorithm
```

using a greedy approach (Lines 14 and 15).

3.5.3 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 the category covered by Algorithm 1. 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. If a vertex doesn't have any neighbours, it will be a part of both the `Min` set and the `Max` set. In Line 26, we make sure such vertices go to only the Max set. In the colouring part, in each iteration, we colour the Max set with a colour c (Lines 8 and 9) and the Min set with $c+1$ (Lines 10 and 11). The algorithm is iterated until there are no vertices left to colour (Line 6). 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:    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                                   ▷ Or V ≠ φ
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|
14:        V ← V \ MaxS
15:        V ← V \ MinS
```

CHAPTER 4

PARALLEL GRAPH COLOURING: INCREMENTAL

4.1 Why Incremental?

Graphs are being used in a varied lot of applications these days. Graphs are ever more important and ever growing. Practical graphs like social networks graphs, communication networks graphs are inherently dynamic. 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.

4.2 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.

4.3 The Two Thread Incremental Model

We assume we have a graph which is already coloured using one of the algorithms, in our case Algorithm 10 from last chapter, on the GPU. We also assume that we have the UCSR graph already residing on the GPU memory.

In the incremental model, when an edge is added, only two vertices are immediately affected. The vertices on which the added edge is incident. If those two vertices have different colours, already, we assume they were coloured optimally by the main algorithm already and leave those two vertices be with their existing colours. If those two vertices have the same colour, then we have a conflict on the addition of the new edge. We are concerned with only those two vertices for the immediate task of resolving the conflict.

We introduce an algorithm called The Two Thread Incremental Model to deal with recolouring the graph. This model makes use of only two threads, each thread for each of the two vertices in conflict, as our immediate objective is to remove the conflict. But, in our model, while removing the conflict, we try to optimize the colours of those two vertices and thus locally maximizing the colour quality. It follows that when the colours of the two vertices are changed in a locally optimized way, there could be positive changes in the other parts of the graph. We will deal with this later in the propagation part.

Our Two Thread Incremental Model is explained in Algorithm 11. From the structure of the problem and our algorithm, it is clear that we actually need only two threads to deal with locally maximizing the colour quality at the region of edge addition. As we have the graph already on the GPU memory, we don't have an issue with memory latency, the biggest hurdle with GPGPU.

This algorithm is meant for addition of a single edge. Two threads are spawned to run the algorithm. Each thread takes charge of one of the vertices on which the new edge is incident. In our algorithm, we assume uv is the added edge and *Thread 0* is in charge of u and *Thread 1* in charge of v. Initially, each thread modifies its portion of the UCSR to accommodate the newly added edge. Then they check if the current colouring of u and v are the same. If not, we are through.

Algorithm 11 The Two Thread Incremental Model

```
1: procedure TwoThreadIncrementalColouring(G(V,E), edge)
2:     Initialization
3:     Let uv be the newly added edge
4:     Thread 0 represents u and thread 1 represents v
5:     Add the new edge information to the UCSR        ▷ Each thread add its neighbour
6:     synchronize
7:     if colour[u] ≠ colour[v] then
8:         return
9:     for i ∈ {u, v} do                                ▷ In parallel by two threads
10:        oldColour ← colour[i]
11:        possibleColour[threadId] ← Smallest possible colour for i
12:        synchronize
13:        if possibleColour[threadId] = possibleColour[1 − threadId] then
14:            if possibleColour[threadId] < oldColour then
15: Label 1:
16:                if threadId= 0 then
17:                    colour[i] ← possibleColour[threadId]
18:                else
19:                    newColour ← Smallest possible colour > possibleColour[threadId] for i
20: Label 2:
21:                    if newColour < oldColour then
22:                        colour[i] ← newColour
23:            else
24: Label 3:
25:                if threadId= 1 then
26:                    colour[i] ← possibleColour[threadId]
27:        else
28: Label 4:
29:            if possibleColour[threadId] < oldColour then
30:                colour[i] ← possibleColour[threadId]
31:            else
32: Label 5:
33:                if possibleColour[threadId] < possibleColour[1 − threadId]
   then
34:                    colour[i] ← possibleColour[threadId]
```

If the current colouring of *u* and *v* are the same, we first try to find out the least colour they can assume considering their neighbourhood in a greedy fashion. This new colouring cannot be their existing colour, as both of them have each other as a neighbour now. Let the current colouring be called p. Let the newly found lease possible colour for each of them be *x* and *y*. Now, we have $x \neq p$ and $y \neq p$. In the case of $x = y$, z be the least colour greater than y that *v* can take in a greedy fashion. We have many cases that can arise in most of which x and y are interchangeable with no side effects. These cases and the actions required are tabulated in Table 4.1.

No	Case	Action	Label No. in Algorithm
1	$x = y < p;\ z < p$	$colour[u] = x;\ colour[v] = z$	1, 2
2	$x = y < p;\ p < z$	$colour[u] = x$	1
3	$p < x = y$	$colour[v] = y$	3
4	$x \neq y;\ x < y < p$	$colour[u] = x;\ colour[v] = y$	4
5	$x \neq y;\ x < p < y$	$colour[u] = x$	4
6	$x \neq y;\ p < x < y$	$colour[u] = x$	5

Table 4.1: The Two Threads Model: Different Cases

Each of the cases given in the table 4.1 are explained using examples in the figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6.

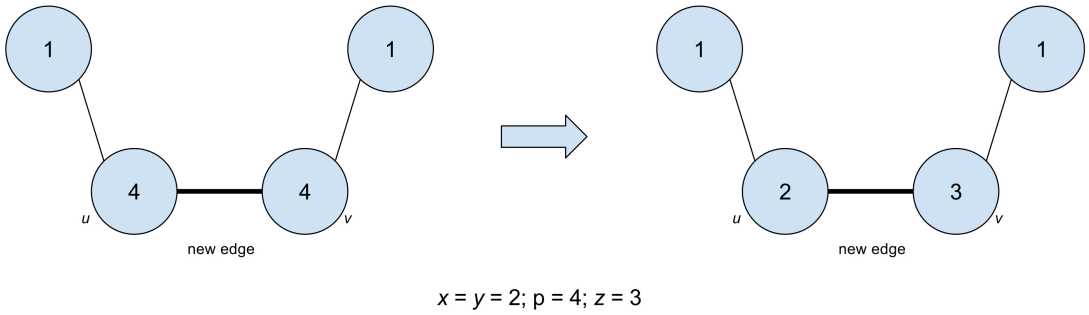

Figure 4.1: **Case 1:** $x = y < p;\ z < p$

4.4 The Many Thread Incremental Model

We have seen in The Two Thread Incremental Model in the last section and how it actually requires only two threads per addition of a new edge. As the graph is already on the GPU, we don't have much memory latency issues too. But then, GPUs are

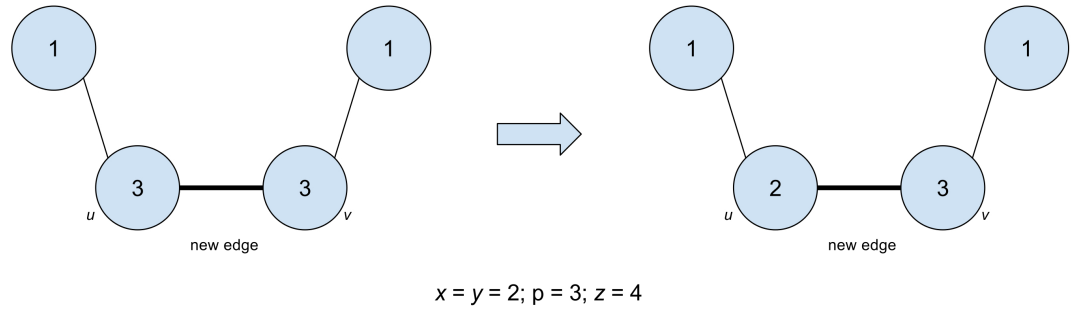

Figure 4.2: **Case 2:** $x = y < p;\ p < z$

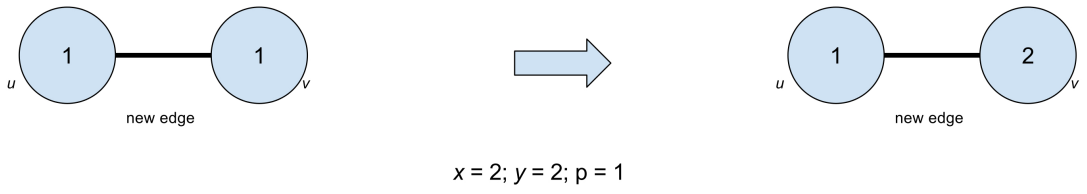

Figure 4.3: **Case 3:** $p < x = y$

devices for massive parallelism. In this section, we will try to increase the parallelism involved in Incremental colouring.

As we have seen in the last session, we can handle a newly added edge with 2 threads. That is, if we colour on the go, we have to call a kernel with 2 threads each time we come across a new edge. To increase parallelism, we can try to club a bunch of new edges and process them together. But these edges shouldn't have any data races among them. So, we try to club edges which won't have any data races amongst them and process them all together in parallel. We call our new approach **The Many Thread Incremental Model** and it is explained in Algorithm 12.

According to our algorithm, we keep on taking new edges as inputs and add them to `edgeSet`, as long as all the edges stored in our `edgeSet` are data race free. That is, there are no two edges which are incident on a common vertex. If a new edge added doesn't hold this true, we process the existing `edgeSet`, all the edges processed in parallel with each edge getting two threads each, using `TwoThreadIncrementalModel` as discussed in the previous section. Once all the edges from `edgeSet` are processed,

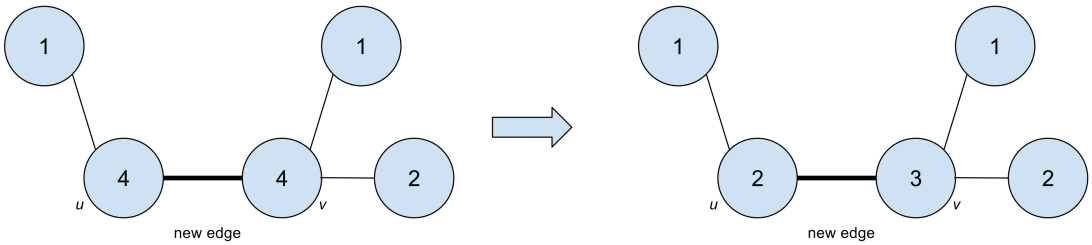

Figure 4.4: **Case 4:** $x \neq y;\ x < y < p$

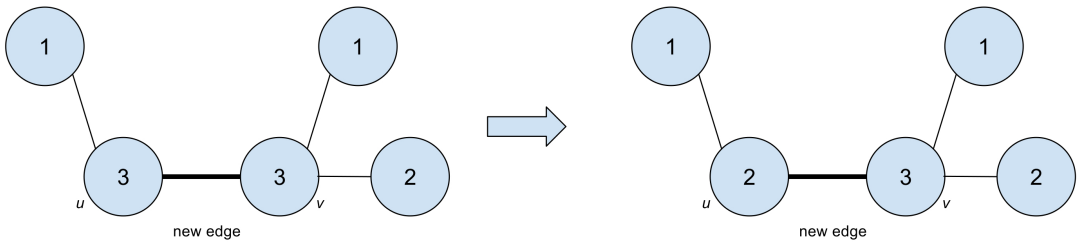

Figure 4.5: **Case 5:** $x \neq y;\ x < p < y$

it is cleared and the new edge is added to it. Then it goes back to accepting more edges.

New added edges can create conflicts. These conflicts will be resolved only after these edges are processed. So, it is not wise to wait for very long to club them with other edges for added parallelism. So, in our implementation, we have set a limit of 1024 edges as the maximum `edgeSet` size before it is forcibly processed.

4.5 Propagation

As discussed earlier, our algorithm tries improves the colouring locally at the end points of the new edge added. But, we consider only those two vertices. If any of them changes its colouring, it could have an effect on its adjacent vertices who could now probably improve their own colouring as they themselves might not be locally optimized. When they indeed change their colours, there could be scope for improvement in their neighbourhood. So, this has a ripple effect. The need for propagation and propagation itself

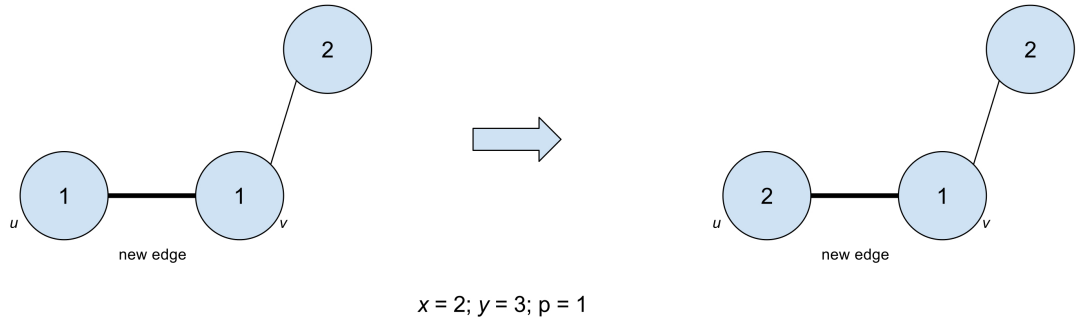

Figure 4.6: **Case 6:** $x \neq y;\ p < x < y$

Algorithm 12 The Many Thread Incremental Model

```
1:  procedure ManyThreadIncrementalColouring(G(V,E), edge)
2:      Initialization
3:      edgeSet ← φ
4:  NewEdge:
5:      Take the new edge to be added, ē, as the input
6:      ū, v̄ ← endpoints of ē
7:      for each e ∈ edgeSet do
8:          u, v ← endpoints of e
9:          setU ← V \ adj(u)                ▷ adj(k) gives adjacent vertices of k
10:         setV ← V \ adj(v)
11:         setPermissible ← setU ∩ setV
12:         if ū ∉ setPermissible OR v̄ ∉ setPermissible then
13:             // Done in parallel for all edges in the edgeSet
14:             TwoThreadIncrementalColouring(G(V,E), edgeSet)
15:             edgeSet ← {ē}
16:             goto NewEdge
17:     edgeSet ← edgeSet ∪ ē
18:     goto NewEdge
```

is portrayed with the help of an example in the figure 4.7.

Though our locally optimizing algorithm works well without considering the ripple effect, it's always good to try for better colour quality. We introduce the idea of propagation to handle this ripple effect. Propagation is done only after a fixed number of edge additions are done so as to save on computation. Our propagation procedure propagates the newly changed colouring info across the graph to the vertices who could possibly be affected.

We discuss our propagation algorithm in the next chapter when we discuss Decremental graph colouring as the propagation step is one and the same for both Incremental

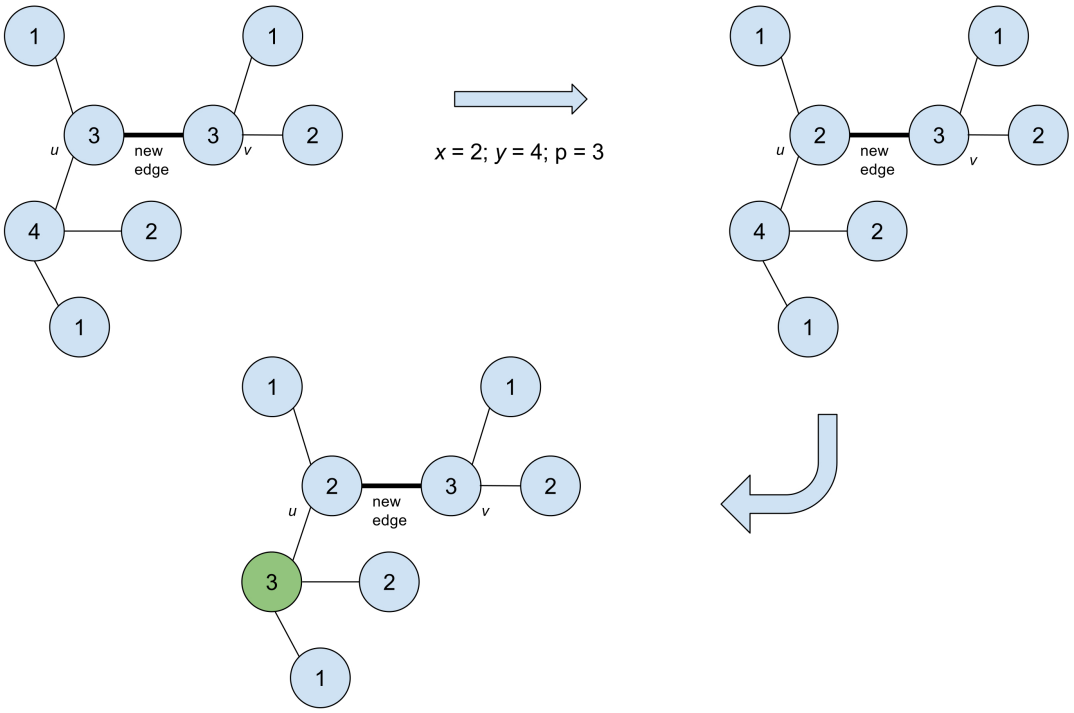

Figure 4.7: 1-step propagation after adding a new edge and processing it.

and Decremental graph colouring.

CHAPTER 5

PARALLEL GRAPH COLOURING: DECREMENTAL

5.1 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 on the graph. Decremental approach helps save running time and avoids unnecessary computations.

5.2 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.

5.3 The Two Thread Decremental Model

We assume that we have the already coloured graph residing in the GPU memory. Now, we decrement an edge, that is, we remove an edge. When an edge is removed, there

is no necessity to change the colouring as it can't inflict any conflict. So, if an edge is removed, we can leave the colouring as it is. But when an edge is removed, there is a possibility that we can improve the existing colouring. And to deal with the same, we introduce our model which we call *The Two Thread Decremental Model*.

In *The Two Thread Decremental Model*, we try to optimize the existing colouring in a localized fashion. As in the case of Incremental colouring discussed in the last chapter, we need only two threads to achieve the same. The scheme for *The Two Thread Decremental Model* is given in Algorithm 13.

Algorithm 13 The Two Thread Decremental Model

```
 1: procedure TWOTHREADDECREMENTALCOLOURING(G(V,E), EDGE)
 2:     Initialization
 3:     Let uv be the newly deleted edge
 4:     Thread 0 represents u and thread 1 represents v
 5:     Remove the deleted edge from the UCSR    ▷ Each thread deletes its neighbour
 6:     synchronize
 7:     for i ∈ {u, v} do                          ▷ In parallel by two threads
 8:         oldColour ← colour[i]
 9:         possibleColour ← Smallest possible colour for i
10:         if possibleColour < oldColour then
11:             colour[i] ← possibleColour
```

The model is pretty simple. Two threads are spawned to run the algorithm. Each thread takes charge of one of the vertices on which the deleted edge was incident. We assume *uv* is the newly deleted edge and *Thread 0* is in charge of *u* and *Thread 1* in charge of *v* (Line 4). Initially, each thread modifies its portion of the UCSR to reflect the deletion of edge *uv* (Line 5). Once the UCSR is updated and synchronized (Line 6), each thread tries to find the smallest colour its vertex can assume in the changed neighbourhood, `possibleColour` (Line 7 to 9). If this smallest colour the vertex can assume, `possibleColour`, is smaller than its current colour, its colour is updated to `possibleColour` (Line 10 to 11).

5.4 The Many Thread Decremental Model

In the previous section we discussed our *Two Thread Decremental Model*. As was discussed, we need only two threads to handle a deleted edge. The issue here is that,

for every edge deleted, we need to spawn a CUDA kernel with just two threads. There are two issues with this. The first issue is that we are not enough use of the massive parallelism that GPUs offer. The second issue is that each kernel call comes with some overhead. There is some cost involved while CPU invokes the GPU for computation. So, it is in our best interest to increase the number of threads used per kernel call and to reduce the number of times kernels are invoked. We present our model we call *The Many Thread Decremental Model* to handle the same.

Algorithm 14 The Many Thread Incremental Model

```
 1: procedure MANYTHREADDECREMENTALCOLOURING(G(V,E), EDGE)
 2:     Initialization
 3:     edgeSet ← φ
 4: Edges To Be Deleted:
 5:     Take the newly deleted edge, ē, as the input
 6:     edgeSet ← edgeSet ∪ ē
 7:     if more edges to be deleted then
 8:         goto Edges To Be Deleted
 9:     // Two threads per edge are spawned
10:     for each e ∈ edgeSet do                          ▷ Done in parallel
11:         Each thread modifies UCSR to remove its corresponding neighbour
```

The scheme for *The Many Thread Decremental Model* is given in Algorithm 14. As deletions of edges don't inflict in colouring conflicts, we can actually choose not to recolour the graph on edge deletions. But the edge deletions should be reflected in the UCSR of the graph residing on the GPU. This we can do in parallel for all the deleted edges together. We can't really locally recolour the vertices on which the deleted edges were incident in parallel, like in Algorithm 13, as this might lead to data races. The edges considered in parallel might not be independent of each other. So, our model chooses not to recolour the graph, atleast not right away, and just deletes the edges from the UCSR of the graph residing on the GPU in parallel (Lines 10 and 11).

But it is clear that all these edge deletion could lead to better colouring of the graph. And to accommodate the same, we introduce propagation to propagate the changed structure of the graph through the graph.

5.5 Propagation

The need for propagation should already be clear from last chapter as well as last section. In Figure 4.7, we showed how propagation could help improve the colouring after addition of edges. If we use Algorithm 13 for edge deletions, then we already should have locally improved colouring. In that case, we can use propagation to possibly improve colour quality. If we use Algorithm 14, we leave the colours as it is. We don't change them. In that case, propagation very likely will improve the colour quality.

In terms of likelihood for improvement in colour quality, we have three approaches to propagation namely the pessimistic approach, the semi-optimistic approach and the optimistic approach.

5.5.1 Pessimistic: No Propagation

In the pessimistic approach, or what we call No Propagation, we leave the colours untouched. That is, we don't do any propagation. After Algorithms 11, 12 and 13, which changes the colouring locally, if we feel that there is meagre chance for the colour quality to improve or if we feel the cost-benefit in terms of time/computation involved *vs* colour quality improved, we can choose not to do propagation in order to save on computation and time.

5.5.2 Semi-Optimistic: One Time Propagation

In the semi-optimistic approach, or what we call One Time Propagation, we propagate the information through the graph in one go. That is, the information is transferred only by one level. The possible ripple effect of propagation is downplayed in this approach. After Algorithms 11, 12 and 13, the immediate neighbours of the locally improved vertices could possibly get better colours. After Algorithm 14, the affected vertices themselves and their immediate neighbours could possibly get better colours.

This approach is a trade off between improvement in colour quality *vs* computation/time. We try to improve the colour quality by doing less computation and hence

supposedly saving on time. We use Algorithm 4 to find Maximal VIS iteratively to propagate the information. The scheme is given in Algorithm **??**.

5.5.3 Optimistic: Wave Propagation

In the optimistic approach, or what we call Wave Propagation, we consider the possible ripple effect of propagation. That is, if a vertex changes its colour, it colour affect the colouring of all its neighbours. So, we process all its neighbours for possible colouring improvements. In case any of such neighbours improves its colouring, then its neighbours could also possibly improve their colouring. This then ripples through the graph until there are no more vertices with possible colouring improvements. This propagation happens in waves of possible colouring improvements and hence we call it Wave Propagation.

This approach might be a little time consuming as we have to keep on adding and removing vertices to and from the process queue. But this approach is highly likely to improve colour quality as it propagates the latest information through the graph. The biggest challenges here are the number of iterations, *waves*, before the algorithm terminates and the maintenance of a concurrent process queue.

CHAPTER 6

EXPERIMENTAL EVALUATION

6.1 Experimental Setup

6.2 Test Data

6.3 Parallel Graph Colouring on GPU

6.4 Incremental Parallel Graph Colouring on GPU

6.5 Decremental Parallel Graph Colouring on GPU

CHAPTER 7

CONCLUSION AND FUTURE WORK

REFERENCES

1. **Amdahl, G. M.**, Validity of the single processor approach to achieving large scale computing capabilities. *In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring). ACM, New York, NY, USA, 1967. URL http://doi.acm.org/10.1145/1465482.1465560.

2. **Appel, K.** and **W. Haken** (1977). Every planar map is four colorable. part i: Discharging. *Illinois J. Math.*, **21**(3), 429–490. URL http://projecteuclid.org/euclid.ijm/1256049011.

3. **Bozdag, D.**, **Ü. V. Çatalyürek**, **A. H. Gebremedhin**, **F. Manne**, **E. G. Boman**, and **F. Özgüner**, A parallel distance-2 graph coloring algorithm for distributed memory computers. *In* **L. T. Yang**, **O. F. Rana**, **B. D. Martino**, and **J. Dongarra** (eds.), *High Performance Computing and Communications, First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005, Proceedings*, volume 3726 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN 3-540-29031-1. URL http://dx.doi.org/10.1007/11557654_90.

4. **Bozdag, D.**, **A. H. Gebremedhin**, **F. Manne**, **E. G. Boman**, and **Ã. V. ÃĞatalyÃijrek** (2008). A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Parallel Distrib. Comput.*, **68**(4), 515–535. URL http://dblp.uni-trier.de/db/journals/jpdc/jpdc68.html#BozdagGMBC08.

5. **Briggs, P.**, **K. D. Cooper**, **K. Kennedy**, and **L. Torczon** (2004). Coloring heuristics for register allocation. *SIGPLAN Not.*, **39**(4), 283–294. ISSN 0362-1340. URL http://doi.acm.org/10.1145/989393.989424.

6. **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.

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

8. **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.

9. **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.

10. **Herzberg, A. M.** and **M. R. Murty** (2007). Sudoku squares and chromatic polynomials. *Notices of the AMS*, **54**(6), 708–717.

11. **Hovland, R. J.** (2008). Latency and bandwidth impact on gpu-systems. *Norwegian University of Science and Technology*.

12. **Jensen, T. R.** and **B. Toft**, *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

13. **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.

14. **Khor, S.** (2009). Application of graph coloring to biological networks. *arXiv preprint arXiv:0912.3461*.

15. **Lakshman, T.**, **A. Bagchi**, and **K. Rastani** (1994). A graph-coloring scheme for scheduling cell transmissions and its photonic implementation. *Communications, IEEE Transactions on*, **42**(234), 2062–2070.

16. **Lee, V. W.**, **C. Kim**, **J. Chhugani**, **M. Deisher**, **D. Kim**, **A. D. Nguyen**, **N. Satish**, **M. Smelyanskiy**, **S. Chennupaty**, **P. Hammarlund**, **R. Singhal**, and **P. Dubey** (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, **38**(3), 451–460. ISSN 0163-5964. URL http://doi.acm.org/10.1145/1816038.1816021.

17. **Leighton, F. T.** (1979). A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, **84**(6), 489–506.

18. **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.

19. **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.

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

21. **NVIDIA Corporation** (2016*b*). curand: Cuda toolkit documentation. URL http://docs.nvidia.com/cuda/curand/#axzz475iTZCEH. [Online; accessed on 22 April, 2016].

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

23. **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.

24. **Thomas, R.** (1998). An update on the four-color theorem. *Notices of the American Mathematical Society*, **45**(7), 848–859.

25. **Welsh, D. J. A.** and **M. B. Powell** (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, **10**(1), 85–86. URL http://comjnl.oxfordjournals.org/content/10/1/85.abstract.

26. **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].

27. **Wilson, R. J.**, *Four Colors Suffice: How the Map Problem Was Solved*. 2014. ISBN 9780691158228 0691158223.

28. **Zuckerman, D.**, Linear degree extractors and the inapproximability of max clique and chromatic number. *In Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, STOC '06. ACM, New York, NY, USA, 2006. ISBN 1-59593-134-1. URL http://doi.acm.org/10.1145/1132516.1132612.