RB-Tree Implementation on GPU

A Project Report

submitted by

ROHITH KUMAR MIRYALA

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

April 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **RB-tree Implementation on GPU**, submitted by **Rohith Miryala**, to the Indian Institute of Technology, Madras, for the award of the degree of **B.tech**, 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.

Prof. Rupesh Nasre
Research Guide
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 24th April 2016

ACKNOWLEDGEMENTS

I am very thankful to my advisor Rupesh Nasre for letting me choose this project. He has been very encouraging and always helped me whenever I was stuck. I also thank Anurag Ingole who helped me in getting familiar with the OpenCL language.

ABSTRACT

Latest works show that GPU's have very high computing capability and they can be used for graph processing. The problem with GPU is dynamically updating graphs. Our goal is to implement RB-tree on GPU and compare the insertion times taken on CPU and GPU.

We used OpenCL and C languages for coding. Since OpenCL doesn't support dynamic allocation of memory on GPU, we prematurely assigned memory required for inserting nodes into RB-Tree.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

GPU is a graphical Processing Unit. Most of the today's GPUs are used for SIMD(Single Instruction Multiple Data) type of operations. A GPU can spawn millions of threads at once and compute. A GPU has atomic operations at hardware level which are very fast. Hence, doing parallel operations on GPU are much faster. The current graph algorithms on GPU are Breadth First Search(BFS), Single Source Shortest Path (SSSP), Minimum Spanning Tree(MST) etc. All of these algorithms process on the given trees but there is no dynamic updating in graphs. Most of the algorithms that exist now use the GPU to do some calculations on it and get back the result. So we wanted to implement RB-Trees on GPU and check how GPU performs.

1.1 Red Black Trees

Red Black Trees are binary search trees with colors attached to the nodes. By maintaining the invariants on node colors, the tree is always balanced. A red black tree has following properties

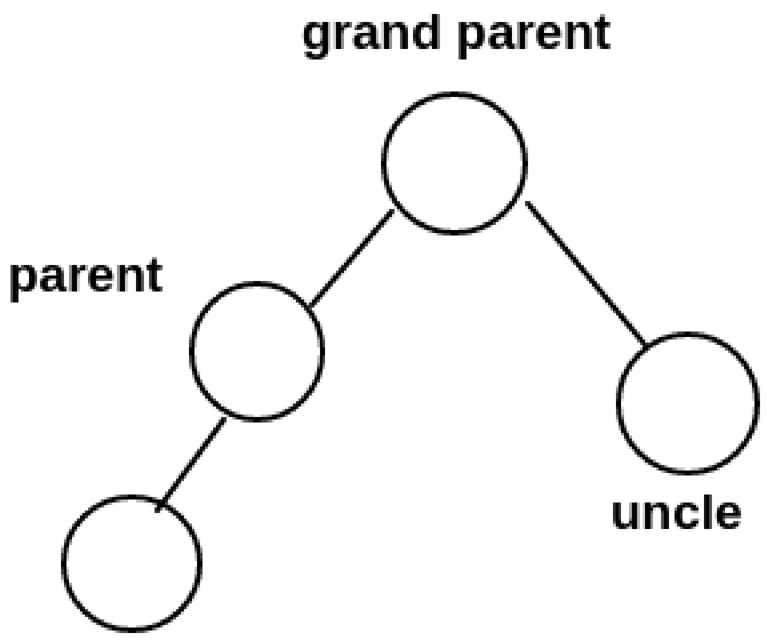
1. Every node is either red or black
2. The root is black
3. Every Leaf(NIL) is black
4. If a node is redd, hen both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

By maintaining this invariants, the tree is always balanced.

Whenever a new node is inserted, it is always colored black. Node insertion is done same as binary search tree. But since the node is colored black, it might violate the properties. So a new fucntion RB-InsertFixup is called which restores the RB-Tree properties.Upon deletion of a node also, RB-DeleteFixup is called to restore RB-tree properties. Insertion, Deletion and Search, all 3 operations take O(log n) time in RB-Trees.

1.2 Proposed Mechanism

Standard algorithms use linked lists to implement RB-Trees. But passing the data to OpenCL kernel in the form of liked list is a difficult task. Also OpenCL does not support dynamic memory allocation. So, it cannot be implemented as linked lists. We cannot directly transfer the tree pointer because OpenCL kernel has different pointers from CPU. So we decided to use arrrays to implement the RB-Trees. The graph is represented using 6 arrays Node, LC(Left Child), RC(Right Child), P(Parent Node), C(Color of the Node), L(Lock on the node).Node array has the value of the node. LC, RC, and P points to index of Node array. Color is either 0(black) or 1(Red).L is used to acquire lock on the node. 0th element is used as sentinel node. It has black color.Since nodes are added in an array, we need to use a flag to track the index of the last node inserted. Also a variable 'R' is used which maintains the index of the root node.
Since memory cannot be dynamically allocated on kernel, we initialise the arrays with extra memory.We take command line arguments to specify the number of nodes initial tree should be made of and the number of nodes to be inserted.A work list of nodes to be inserted is created on CPU and sent to kernel.
The kernel is launched with threads as many as nodes inserted. Since nodes are inserted parallel, we have to acquire locks to prevent data corruption. We acquire locks on current node, its parent, grand parent and uncle and do operations.



if((P[index] == LC[P[P[index]]]) L[P[index]]==0 L[P[P[index]]]==0 L[RC[P[P[index]]]]==0){

$$l2 = atomic\_cmpxchg(L[min], 0, 1); \tag{1.1}$$

$$l3 = atomic\_cmpxchg(L[mid], 0, 1); \tag{1.2}$$

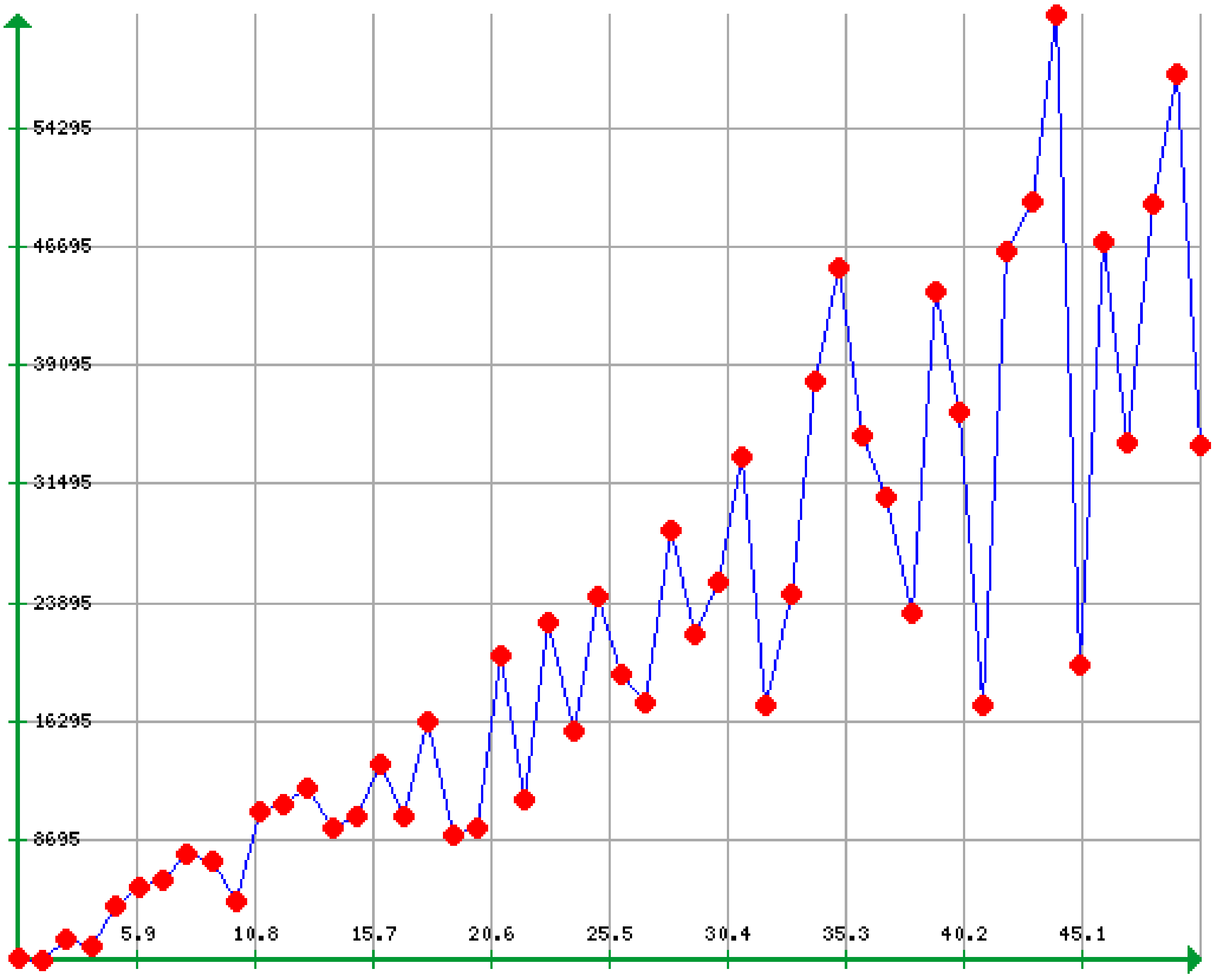
$$l4 = atomic\_cmpxchg(L[max], 0, 1); \tag{1.3}$$

First locks are checked normally then we use atomic operations. Because atomic operations are expensive, we don't bother using atomics once we know that a node is already locked. This is an optimisation done on locks.(See appendix for full code) Every thread tries to acquire locks on four nodes and if it couldn't get the lock, the thread is stopped and the kernel is launched again in next iteration and the thread continues where it stopped. The locks are acquired in the order of the node indexes to prevent live locks.

Live locks can occur one one thread gets lock on one node and the other thread acquires lock on other node, since both nodes doesn't have locks on four nodes, both threads are suspended and same repeats in next iteration. This can be prevented by acquiring locks in the order of indices.

1.3 Implementation Results

Initially RB-Tree is implemented on CPU.A tree is created with 2 lakh nodes. A graph is plotted against the time taken and percentage of nodes inserted.

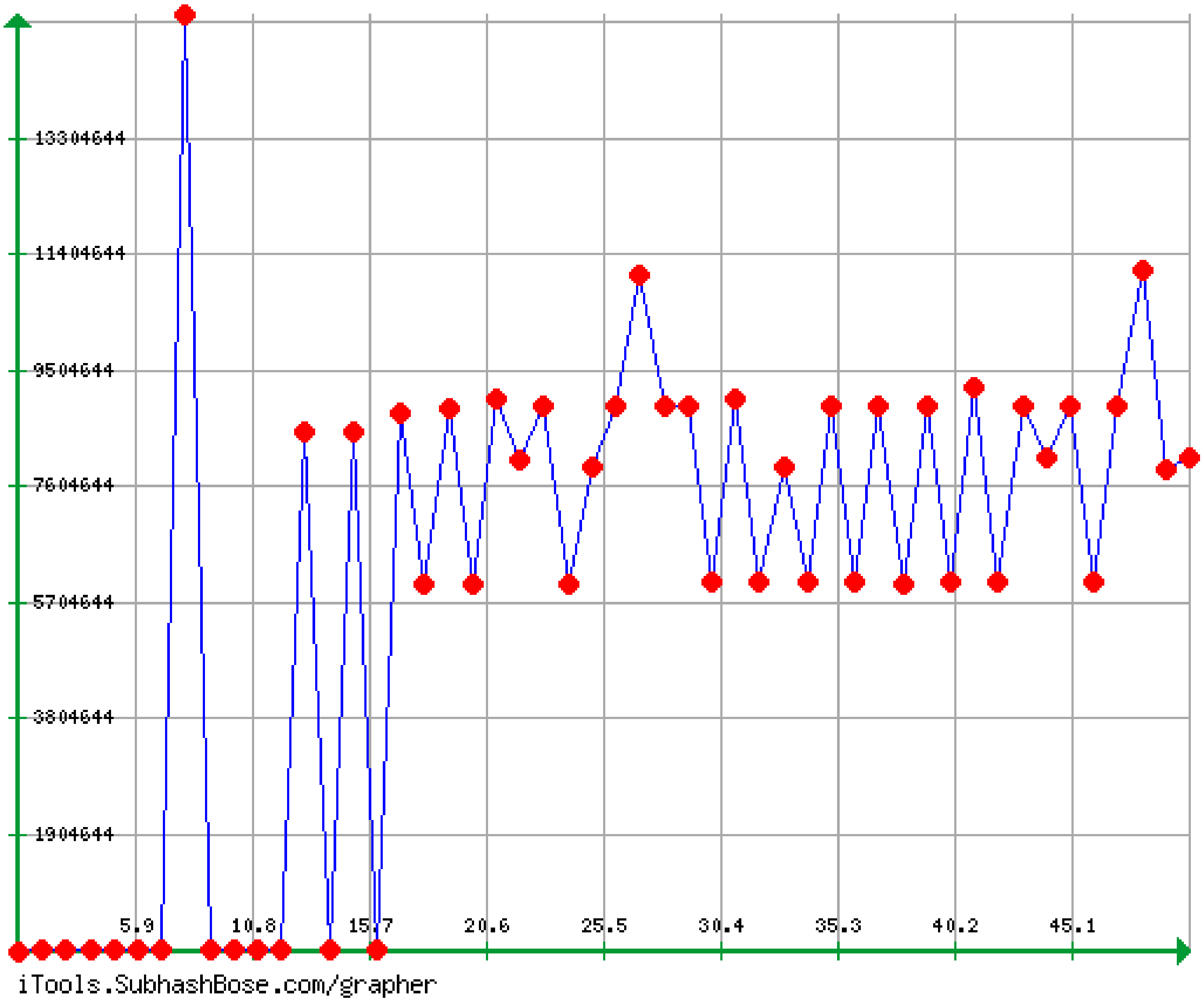


X-axis is percentage of nodes added

Y-axis is time taken in micro seconds.

The graph is pretty expected since nodes are added one by one, as number of nodes increases time taken increases linearly. Some points have less time because it might have involved less number of rotations in that particular case.

Next the tree is implemented on GPU and similar graph is plotted.This time graph is unexpected.There is a sudden rise in the time taken on GPU at 10-15%. The reason is not known yet.



1.4 Future Work

At present, only insertion is done. Deletion has to be implemented. In the present implementation a single node is used as sentinel node. Because of this many threads which have only one sentinel node in common cannot do operations simultaneously.This decreases the performance. So different sentinel nodes can be used and performance can be checked. Since we are getting a step at 10-15%of nodes, we can insert nodes by sending them in batches of 10% of nodes.

APPENDIX A

RB-InsertFixup

```
while(C[P[index]]){
    if( (P[index] == LC[P[P[index]]]) && L[P[index]]==0 && L[P[P[
   index]]]==0 && L[RC[P[P[index]]]]==0 ){
      __private int l1,l2,l3,l4;
      //l1 = atomic_cmpxchg(&L[index],0,1);
      a = P[index];
      b = P[P[index]];
      c = RC[P[P[index]]];
      if(a>b && a>c){
        max = a;
        if(b>c){
          mid = b;
          min = c;
        }
        else{
          mid = c;
          min = b;
        }
      }
      else if(a<b && a<c){
        min = a;
        if(b>c){
          max = b;
          mid = c;
        }
        else {
          max = c;
          mid = b;
        }
      }
      else {
        mid = a;
        if(b>c){
          max = b;
          min = c;
        }
        else{
          max = c;
          min = b;
        }
      }
      l2 = atomic_cmpxchg(&L[min],0,1); //take locks in increasing
    order of node indexes
      l3 = atomic_cmpxchg(&L[mid],0,1);
      l4 = atomic_cmpxchg(&L[max],0,1);
      //L[index]=1; L[P[index]]=1;  L[P[P[index]]]=1; L[RC[P[P[index
    ]]]]=1;
      if(!l2 && !l3 && !l4){
        __private int z = c;
        if(C[z]){
          C[P[index]] = 0;
          C[z] = 0;
          C[P[P[index]]] = 1;
          __private int temp;
          temp = index;
          index = P[P[index]];
          L[temp]=0; L[P[temp]]=0; L[RC[P[P[temp]]]]=0;
        }
        else {
          if(index == RC[P[index]]){
            index = P[index];
            leftRotate(RC,LC,P,index,R);
          }
          C[P[index]] = 0;
          C[P[P[index]]] = 1;
          rightRotate(RC,LC,P,P[P[index]],R);
          L[index]=0; L[P[index]]=0;  L[RC[P[index]]]=0; L[RC[RC[P[
    index]]]]=0;
          return 0;
        }
      }
      else{
        if(l2==0)L[min] = 0; //release locks acquired by this thread
    because it cant enter critical section
```

```
        if(l3==0)L[mid] = 0;
        if(l4==0)L[max] = 0;
        TL[base] = index;
        return 2;
      }
    }
    else if( L[P[index]]==0 && L[P[P[index]]]==0 && L[LC[P[P[index
   ]]]]==0 ){
      __private int l1,l2,l3,l4;
      //l1 = atomic_cmpxchg(&L[index],0,1);
      a = P[index];
      b = P[P[index]];
      c = LC[P[P[index]]];
      if(a>b && a>c){
        max = a;
        if(b>c){
          mid = b;
          min = c;
        }
        else{
          mid = c;
          min = b;
        }
      }
      else if(a<b && a<c){
        min = a;
        if(b>c){
          max = b;
          mid = c;
        }
        else {
          max = c;
          mid = b;
        }
      }
      else {
        mid = a;
        if(b>c){
          max = b;
          min = c;
        }
        else{
          max = c;
          min = b;
        }
      }
      l2 = atomic_cmpxchg(&L[min],0,1);
      l3 = atomic_cmpxchg(&L[mid],0,1);
      l4 = atomic_cmpxchg(&L[max],0,1);
      //L[index]=1; L[P[index]]=1;  L[P[P[index]]]=1; L[LC[P[P[index
    ]]]]=1;
      if( !l2 && !l3 && !l4){
        __private int z = c;
        if(C[z]){
          C[P[index]] = 0;
          C[z] = 0;
          C[P[P[index]]] = 1;
          __private int temp;
          temp = index;
          index = P[P[index]];
          L[temp]=0; L[P[temp]]=0; L[LC[P[P[temp]]]]=0;
        }
        else {
          if(index == LC[P[index]]){
            index = P[index];
            rightRotate(RC,LC,P,index,R);
          }
          C[P[index]] = 0;
          C[P[P[index]]] = 1;
          leftRotate(RC,LC,P,P[P[index]],R);
          L[index]=0; L[P[index]]=0;  L[LC[P[index]]]=0; L[LC[LC[P[
    index]]]]=0;
          return 0;
        }
      }
      else{
        if(l2==0)L[min] = 0;
        if(l3==0)L[mid] = 0;
        if(l4==0)L[max] = 0;
        TL[base] = index;
```

```
            return 2;
         }
      }
}
C[R[0]] = 0;
L[index] = 0;
return 0;
```

A.1 References

1. OpenCL learning:
   https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/contents/

2. Vincent Gramoli, More than you ever wanted to know about synchronisation
   http://dl.acm.org/citation.cfm?id=2688501

3. Introduction to Algorithms, Thomas H.Coremen, Charles E. Leiserson, Ronald R. Rivest, Clifford Stein

4. Large-Scale Graph Processing Algorithms on the GPU
   http://www.idav.ucdavis.edu/ yzhwang/gpugraph.pdf