Concurrent AVL Tree Operations on GPUs

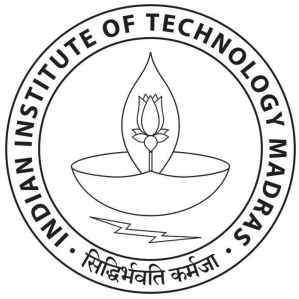
A Project Report

submitted by

ABHISHEK YADAV

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

DUAL DEGREE
BACHELOR AND MASTER OF TECHNOLOGY



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

APRIL 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **Concurrent AVL Tree Operations on GPUs**, submitted by **Abhishek Yadav**, to the Indian Institute of Technology, Madras, for the award of the degree of **B.Tech + M.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.

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

Place: Chennai

Date: 10th March 2017

ACKNOWLEDGEMENTS

I am very thankful to my advisor **Dr. Rupesh Nasre** for letting me work with him on this project and guiding me through the completion of it. He has been kind enough to let me switch the implementation framework from **OpenCl** to **Cuda** owing to lack of proper documentation and online support available for the former. He has been quite encouraging and helped me through his suggestions and invaluable inputs for the project.

I am indebted to IIT Madras and Department of Computer Science and Engineering for providing me the opportunities I never imagined and being the place of best of the times of my life. I would like to thank my friends and professors for their support throughout without whom I wouldn't be the person I am.

ABSTRACT

Recent developments in GPU architecture has prompted researchers to use GPUs to improve the performance of the algorithms which can be parallelized. Several algorithms to perform concurrent insertions, deletions and searches on AVL Tree have been published. However, these algorithms have been implemented to run on multicore CPUs which are optimized for sequential execution and can launch very few number of threads as compared to GPUs. Since general purpose GPUs consist thousands of smaller cores optimized for parallel processing these can be used to execute parallel tasks over multicore CPUs. Update operations are bottlenecks in the performance of AVL trees in concurrent environment. My work is inspired from the work in (7) which introduces the novel concept of Logical Ordering of the tree elements in order to perform concurrent lock-free searches along with update operations.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Binary Search Trees

Binary Search Trees are excellent data structures to implement maps, databases, symbol tables, sets and similar such interfaces. But they are efficient only when they are balanced because straightforward insertions, in the worst case, can lead to highly unbalanced trees which will result in inefficient search(which is required for both deletion and insertion operations) complexity. The solution to this problem is to dynamically rebalance the tree during insert/search/delete operations without destroying the ordering invariant. These type of self-balancing binary search trees have some property which must hold for each node after every operation. Self-balancing property of Balanced Binary Search Trees restrains their heights to O($\log(n)$) which results in efficient height-dependent tree operations and thereby making such structures desirable for applications which require efficient access to their data.

1.2 AVL Trees

AVL Tree, named named after its inventors Adel'son-Vel'skii and Landis is balanced binary search tree. AVL trees maintain *height invariant* property which means, for any internal node, the heights of its left and right subtrees can differ by at most one. After insert and delete operations, *height invariant* property may not hold for some nodes and thus requires rebalancing operations to be performed to restore the height-invariant property of the tree. For the convenience of representation, we define left child, right child, height and parent of a node n as **left**(n), **right**(n), **height**(n) and **parent**(n) respectively. Some more notations will be introduced later to represent certain properties

of the nodes during concurrent operations. Height of any node, n, is defined as:

$$\textbf{height}(n) = \begin{cases} -1, & \text{if n=NULL} \\ 1 + \textbf{max}(\textbf{height}(\textbf{left}(n)), \textbf{height}(\textbf{right}(n)), & \text{otherwise} \end{cases}$$

Balance factor, **bal**(n) of a node n, is defined as the difference in heights of the sub-trees rooted at n. Some authors define it as **bal**(n) = **height**(**left**(n)) - **height**(**right**(n)) whereas others define it as **bal**(n) = **height**(**right**(n)) - **height**(**left**(n)). I have used the first definition of balance factor throughout. Thus, a Binary Search Tree is an AVL tree if, for each node,n, of the tree, **bal**(n) $\in \{-1, 0, 1\}$.

1.3 AVL Tree Rotations

In an AVL tree, after performing insertion and deletion operations heights of the nodes in the path of insertion/deletion gets changed. Consequently, the **balance factor** of some of the nodes on the path of insertion/deletion also gets changed. If all the nodes on the path of insertion/deletion satisfy the balance factor condition then operations conclude the same way as in typical binary search tree otherwise rotation operations are performed on one or more nodes to maintain the height invariant or balance factor of the tree. We define critical node, **cn** as the node in the path of insertion/deletion whose balance factor may not satisfy the balance-factor invariant of the avl tree. If balance factor of the critical node does not satisfy the balance-factor property of the avl tree, depending on the direction of insertion/deletion w.r.t. critical node, one of the following four rotation operations are performed to maintain the balance-factor invariant of the affected nodes:

1.3.1 Single Left Rotation

If a node is inserted to the right of the right child of the critical node, **cn**, resulting in the violation of height-invariant property of **cn**, a single **left-rotation** is used to balance the tree. Let t=**right**(cn), following changes take place as a result of the rotation-

- **parent**(t) $\leftarrow$ **parent**(cn)
- **parent**(cn) $\leftarrow$ t

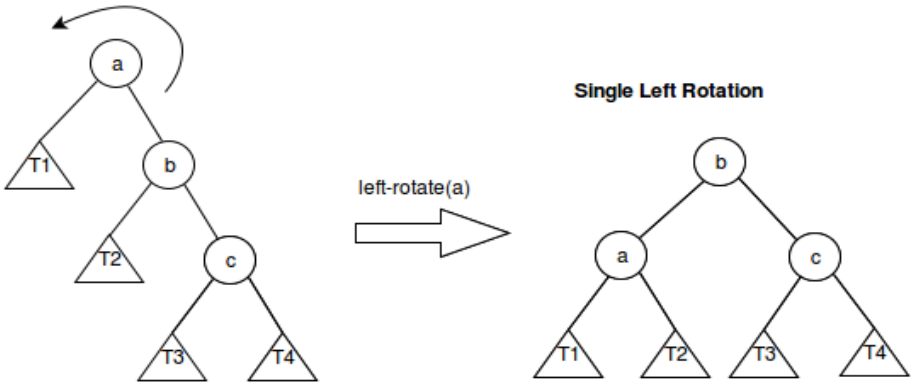


Figure 1.1: Single Left Rotation

- **right**(cn) ← **left**(t)
- **parent**(**left**(t)) ← cn
- **left**(t) = cn
- **left**(**parent**(t)) ← t or **right**(**parent**(t)) ← t depending on whether cn was the left or the right child of its parent.

Heights of **cn**, **t** and **parent**(t) are updated based on the heights of the subtrees added to the left/right of them. In figure 1.1, node **a** is a critical node whose **balance-factor** property is destroyed by an insertion in the right-subtree of its right-child **b** which is restored by a single left rotation at **a**.

1.3.2 Single Right Rotation

If a node is inserted to the left of the left child of the critical node,**cn**, a single right-rotation at **cn** balances the tree. Let t= **left**(cn) and p=**parent**(cn), following changes take place due to rotation at **cn**-

- **parent**(t) ← **parent**(cn)
- **parent**(cn) ← t
- **left**(cn) ← **right**(t)
- **parent**(**right**(t)) ← cn
- **right**(t) = cn

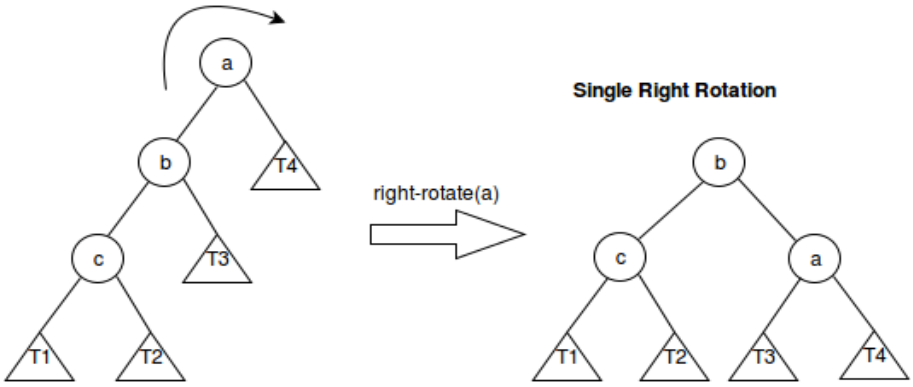


Figure 1.2: Single Right Rotation

Assuming that the **cn** was the left-child of its parent before rotation, heights of the nodes affected after this rotation are updated as follows:

$$\textbf{height}(cn) = 1 + \max(\textbf{height}(\textbf{left}(cn)), \textbf{height}(\textbf{right}(cn)))$$

$$\textbf{height}(t) = 1 + \max(\textbf{height}(cn), \textbf{height}(\textbf{left}(t)))$$

$$\textbf{height}(p) = 1 + \max(\textbf{height}(t), \textbf{height}(\textbf{right}(p)))$$

In Fig. 1.2, right rotation takes place at node **a** which is the critical node.

Double Rotation - Left Right

If the insertion of a key to the right of the left-child of the critical node, **cn**, results in the violation of height-invariant property at cn, then a double rotation is required to restore the invariant. First, a left rotation is performed at the left child of the critical node followed by a right rotation at the critical node balancing the subtree below critical node. Assuming that before rotation, parent of **cn** is p=**parent**(cn), the left child of the critical node is **l** = **left**(cn) and the right child of the **l** is **r**, following changes take place in the structure of the tree

- **parent**(r) ← p
- **parent**(cn)← r
- **parent**(t) ← r
- **right**(t) ← **left**(r) and **parent**(**left**(r)) ← t
- **left**(cn) ← **right**(r) and **parent**(**right**(r)) ← cn
- **left**(r) ← t and **right**(r) ← cn

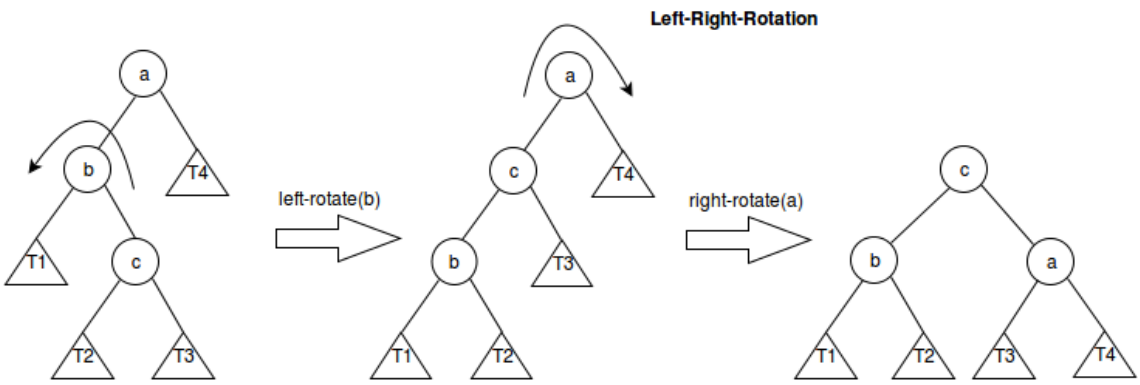


Figure 1.3: Double Rotation - Left Right

Depending on whether **cn** is the left or right child of **p** before rotation, **r** becomes the left or right child of **p** after rotation. Heights of **cn**, **r** and **t** are updated after rotation. Fig. 1.3, shows an example of **Left Right Rotation** where a node is inserted to the right of **b** causing imbalance at critical node **a** which is fixed by a left rotation at b followed by a right rotation at **a**.

Double Rotation - Right Left

If *balance-factor* of the critical node,**cn**, gets destroyed due to insertion of a key to the left of its right child, a **double rotation - Right Left** is required to fix the *balance-factor* property of the nodes in the path of insertion. In the first step of the double rotation, a **Single Right Rotation** is performed at the right child of **cn** followed by a *Single Left Rotation* at *cn* to fix the *avl-tree-invariant* of the nodes in the insertion-path. Assuming that p=**parent**(cn), t= **right**(cn) and l=**left**(**right**(cn)) before rotation, following changes take place after double rotation - Right Left:

- **parent**(l) $\leftarrow$ p and **left**(p) $\leftarrow$ l or **right**(p) $\leftarrow$ l, depending on whether *cn* was the left or right child of its parent before rotation
- **parent**(cn) $\leftarrow$ l
- **right**(cn) $\leftarrow$ **left**(l) and **left**(l) $\leftarrow$ cn
- **parent**(t) $\leftarrow$ l
- **left**(t) $\leftarrow$ **right**(l) and **right**(l) $\leftarrow$ t

Heights of the nodes **t**, **l** and **cn** are updated in after rotation. In Fig. 1.4, **a** is the critical node whose balance-factor is destroyed by an insertion in the left-subtree of its

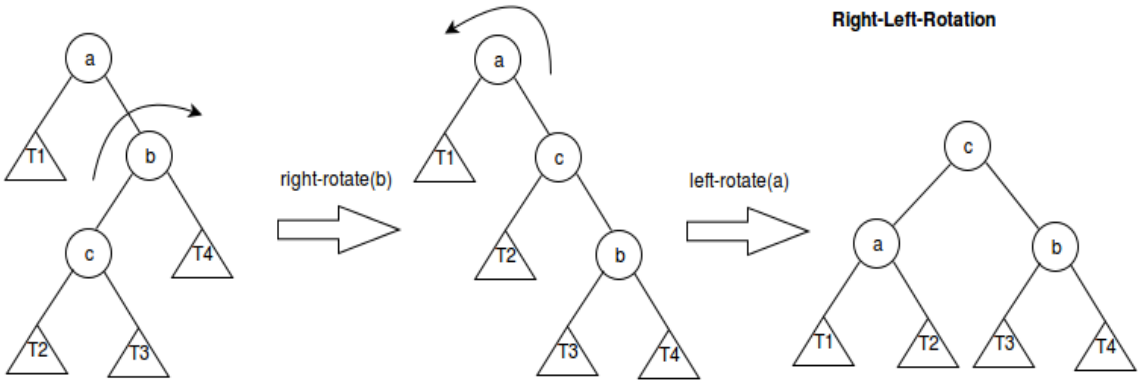


Figure 1.4: Double Rotation - Right Left

right child b, Resulting in a *Double Rotation - Right Left* which balances the tree by restoring the **balance-factor** property of **a,b** and **c** and makes c the parent of both a and b.

1.4 Operations on AVL Tree

1.4.1 Search

Search operation in AVL Tree is performed the same way as it is done in standard Binary Search Tree. Starting at root, search-key is compared with the key of each node, n, it encounters with and returns *true* or n if the *search-key* matches the key of **n**, goes to the left-child of **n** if *search-key* $\leq$ **key**(n) or right-child of **n** if *search-key* $>$ **key**(n). If search-key is not present in the tree, it returns false or the node which would be the parent of the node containing search-key if it were to be inserted, i.e. the node where search ends without success.

Algorithm 1 Search in AVL Tree

```
1:  function SEARCH(root, key′)
2:      if root == NULL then
3:          return root
4:      temp = root
5:      while true do
6:          if key(temp) == key′ then
7:              return temp
8:          child = key′ < key(temp) ? left(temp) : right(temp)
9:          if child == NULL then
10:             return temp
11:         temp= child
```

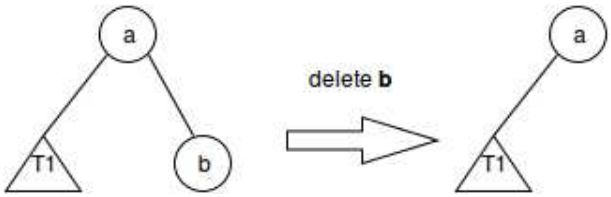


Figure 1.5: Deletion of a leaf node

1.4.2 Insertion

First, a new node is created with the key to be inserted. If root of the tree is **null**, the newly created node becomes the root of the tree and operation concludes, otherwise standard BST operation is performed to find the right location for the newly created node and inserts it. Next, path from newly inserted node to the root is traversed and necessary rotations are performed on the nodes whose balance factor gets disturbed as a consequence of insertion.

Algorithm 2 Insertion in AVL Tree

```
1:  function INSERT(root, key'):
2:      node= Search(root, key')
3:      if node == NULL then
4:          return create_node(key')
5:      else if key(node)==key' then
6:          return node
7:      new_node = create_node(key')
8:      if key(node) > key' then
9:          left(node) ← new_node
10:     else
11:         right(node) ← new_node
12:     parent(new_node) ← node
13:     rebalance(root,node,key')
```

1.4.3 Deletion

Like *Search* and *Insert*, deletion in AVL tree is performed the same way as is done in standard Binary Search Trees. Three types of situation may arise when a node is deleted from BST which are following:

1. If the node to be deleted is the leaf node, as shown in Fig. 1.5, it is physically deleted from the tree and path from its parent to root is traversed to balance the tree.

2. If the node to be deleted has a single child, it is deleted physically while new links between its child and parent are formed. Parent to root path is traversed to

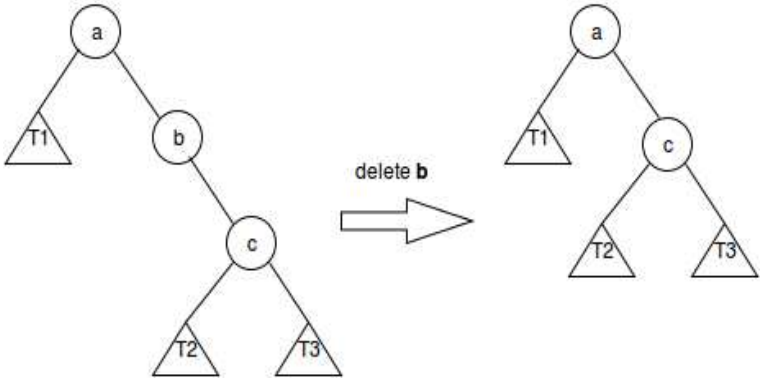


Figure 1.6: Deletion of a node with a single child

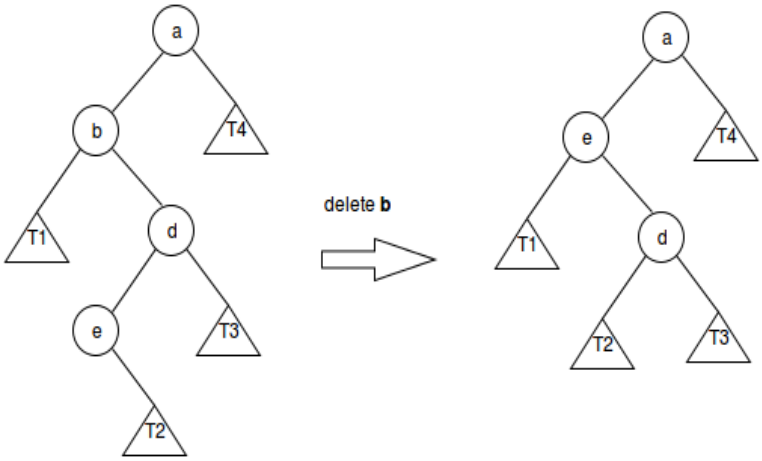


Figure 1.7: Deletion of a node with two children

rebalance the tree. As shown in Fig. 1.6, node **b** with a single child **c** is deleted and consequently c becomes the new child of **a**.

3. If the node to be deleted, **n**, has two children, first **inorder-successor** or **inorder-predecessor** of **n** is obtained (**Note:** I've considered inorder-successor in my implementation to replace the key of the node to be deleted). Next, key of **n** is replaced by the key of **succ**(n) and inorder-successor node is physically deleted. Since, inorder-successor of a node is the node with the smallest key in its right-subtree, which means that either it will have its right child or none; and hence its deletion becomes the deletion of the node with 1 or no child. As shown in Fig. 1.7, request to delete the node with key **b** comes which is replaced by its inorder-successor, **e** which gets physically deleted from the tree.

Algorithm 3 Deletion in AVL Tree

```
1: function DELETE(root, key′):
2:     del_node= search(root, key′)
3:     if del_node == NULL then
4:         return
5:     p= parent(del_node)
6:     if left(del_node) == NULL or right(del_node)==NULL then
7:         child= NULL
8:         if left(del_node) ≠ NULL then
9:             child= left(del_node)
10:        else
11:            child= right(del_node)
12:        if del_node == left(p) then
13:            left(p) = child
14:        else
15:            right(p) = child
16:        if child ≠ NULL then
17:            parent(child)= p
           rebalance(root,p)
18:    else
19:        successor= inorder_successor(del_node)
20:        key(del_node) = key(successor)
21:        p'= parent(successor)
22:        if successor == left(p') then
23:            left(p') = right(successor)      ▷ since successor cannot have left child
24:        else
25:            right(p') = right(successor)
26:        if right(successor) ≠ NULL then
27:            parent(right(successor)) = p'
           rebalance(root,p')
```

CHAPTER 2

Concurrent Operations

AVL trees have been used in databases, symbol tables, dictionaries and many more such interfaces. Enhancement in the computing power of multi-core processors and use of GPUs to perform parallel tasks using thousands of small cores has shifted the focus of the research on the operations of data structures to develop more scalable and efficient algorithms in order to extract parallelism. Since updates(insertion/deletion) in AVL trees require locking the path from the node to be inserted/deleted to the root, these become bottleneck to the performance. Several algorithms have been proposed to avoid deadlock while concurrently performing these blocking operations along with search operation but most of them have been implemented to run on CPUs. General Purpose GPUs(GPGPUs) contain significant number of smaller cores as compared to CPUs and offer better performance if tasks can be parallelized. The main challenge in designing scalable and efficient concurrent BST algorithms is to devise a scalable design for the *search* operation as it is invoked by all three operations. In concurrent execution environment, location of the element being searched may change due to other concurrent operations and elements present in the tree may not be found in it as can be seen in Fig. 2.1.

Several algorithms have been proposed for concurrent operations on AVL trees. In [3], authors have used the concept of virtual plane waves combined with the local information of the nodes such as *dynamic balance factor*(dbal), *dynamic-height*(dheight), virtual depths and so on to derive the relaxed version of the AVL tree locally, and concurrently combine them as the wave moves up, to balance the entire tree. In [5] three types of locks have been used to perform concurrent search and insertion operations but algorithm for concurrent deletion has not been discussed. In [7], Authors have proposed a novel method to perform concurrent lock-free lookup operation along with mutating operations by incorporating logical layout of the tree elements. In [2], mechanisms to perform all three operations concurrently on modified version of AVL trees, Expanded AVL trees, using five locks, and incorporating few more fields to facilitate locking, have

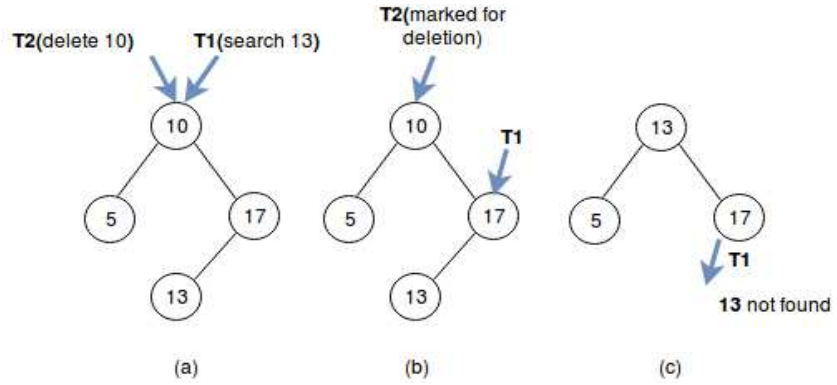


Figure 2.1: Concurrent Search with mutating operations. In (a) thread **T1** starts search for 13 whereas **T2** starts with searching for the node with key 10 for deletion. In (b), **T1** descends down and **T2** deletes 10 by replacing it with its successor 13 and deletes 13 physically from the tree. In (c), 13 is physically moved at root and **T1** fails

been proposed. My GPU based implementation is motivated from the ideas proposed in [7]

2.1 Logical Ordering

A binary search tree stores elements in such a way that a total order can be maintained between them. One such ordering can be obtained by *inorder traversal* of the tree which results in non-decreasing ordering of elements. For instance, inorder traversal of the BST in Fig 2.1(a) results in {5,10,13,17}. To avoid edge cases, we add -∞ and ∞ as smallest and largest elements present in the tree which yields the ordering {-∞, 5, 10, 13, 17, ∞}. Logical ordering of tree-elements can be maintained by storing *inorder-successor* and *inorder-predecessor* information for each node. This enables accessing the next/previous element in the ordering layout through a single pointer instead of traversing several pointers along the physical layout of the tree. As logical ordering remains stable despite mutating operations, search operation can proceed concurrently with mutating operations. Maintaining ordering information results in space overhead but saves time while accessing successors or predecessors required by *delete operation*. It also represents some time overhead as it requires updating the ordering information when it changes, but allows search operations to proceed without any synchronization over the tree layout.

To show how concurrent lock-free search operation can be performed, we consider

the example in fig 2.1. Initial ordering of the tree elements is {-∞, 5, 10, 13, 17, ∞}. After T2 deletes 10 from the tree, ordering is updated to {-∞, 5, 13, 17, ∞}. Earlier when T1 was searching for 13, it failed to find 13 as it was moved to root due to concurrent deletion of 10. However with logical ordering in place, after T1 reaches at 17, as 13 < 17, it finds that the left child of 17 is NULL, it looks up the predecessor of 17 and finds that 13 is present in the tree and successfully terminates.

2.2 Concurrent Operations using Logical Ordering

To maintain the logical ordering of the key elements of the tree, each node is extended with two pointers, **succ** and **pred** to store the address of the successor and the predecessor respectively. By using *succ* pointer, successor of a node can be obtained following a single pointer which, otherwise, would require traversing the path from node to successor which could result in the thread being locked for a longer period of time due to concurrent updates on the path. From here onwards, we denote a node with key k as N_k

Updating *succ* and *pred* fields upon insert and delete: When a node N_k is inserted in the tree, it either becomes left or right child of its parent p. When it becomes left child of p, p becomes it's successor, predecessor otherwise. When it is inserted to the left of p, following changes occur:

- p.**pred**.**succ** = N_k
- N_k.**succ** = p
- N_k.**pred** = p.**pred**
- p.**pred**= N_k

following changes occur when N_k is inserted to the right of p:

- p.**succ**.**pred** = N_k
- N_k.**succ** = p.**succ**
- N_k.**pred** = p
- p.**succ** = N_k

Fig 2.2 shows the changes that occur after a node is inserted.

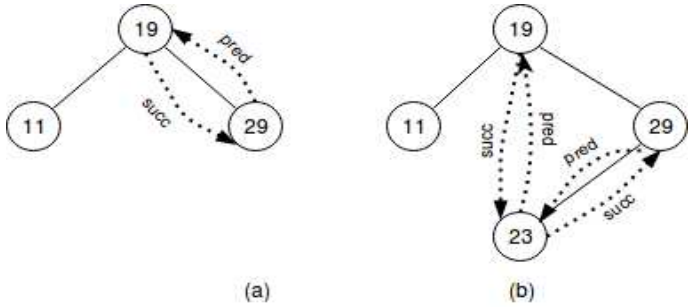


Figure 2.2: Updating successor and predecessor pointer after an insert

upon deletion, irrespective of the number of children the deleted node has, its successor and predecessor are set to point to each other.

2.3 Algorithm

Node structure is modified to accommodate the additional fields incorporated for correct operations in concurrent environment. Node contains *mark* field to indicate if node is deleted; initially, it is set to **false**. Whereas *tree_lock* protects tree's physical layout fields, *succ_lock* protects tree's logical ordering fields, *succ* and *pred*. Initially, tree contains two nodes with keys -∞ and ∞, both being predecessor and successor of each other. Node N_{∞} is made the root node and $N_{-\infty}$ the sentinel node which can only be accessed via *pred* pointer in tree's logical ordering layout.

```
struct Node<T>{
    T key;
    int height;
    struct Node<T> left,right;
    struct Node<T> parent;
    struct Node<T> succ,pred;
    Lock succ_lock, tree_lock;
    bool mark;
}
```

2.3.1 Search

The *search* operation is performed the same way as in sequential implementation. As *Search* does not acquire locks on tree's physical layout fields and doesn't restart, it is oblivious to the location updates caused by other concurrent operations. Some elements, k, despite being present in the tree, may not be returned by *search* due to mutating operations as shown in fig. 2.1. In such cases, we determine two nodes N_{k1} and N_{k2} in the tree such that $k \in (k1, k2)$. To find N_{k1}, *pred* pointers are traversed, starting from the node returned by *search*, until a node with key $\leq k$ is found; similarly, N_{k2} is obtained by traversing *succ* pointers until a node with key $\geq k$ is found. Finally, k is compared with the key returned by traversing *succ* pointers in the logical ordering layout of the tree and terminates successfully if equal, fails otherwise.

Algorithm 4 Search in concurrent environment

```
1: function CONCURRENT_SEARCH(k):
2:     node= search(k);
3:     while k < node.key do
4:         node= node.pred;
5:     while k > node.key do
6:         node= node.succ
7:     return (node.key ==k && !node.mark)
```

2.3.2 Insert

Insert(k) operation starts with by calling **search**(k) which returns a node n. If update operations don't modify n after it was returned by search, n can be a node with key k or a node containing successor of k or (iii) a node containing predecessor of k. Next, successor and predecessor of k in the tree, namely **s** and **p**, are determined and **p** is locked using *succ_lock*. If $k \in$ (**p**, **s**] and **p** is not marked it moves to next step, otherwise operation restarts as update operations had modified the location of insert and **p** is unlocked. If **s** contains key equal to k operations fails, releases lock on **p** and terminates; otherwise successful insertion begins by choosing the correct parent between **s** and **p** and locking it before updating the logical ordering followed by physical layout fields of the tree. Finally, rebalancing operation is applied, if required, to balance the imbalanced nodes.

Algorithm 5 Concurrent Insert in AVL Trees

```
1:  function CONCURRENT_INSERT(k)
2:      while true do
3:          n = search(k)
4:          p = n.key < k ? n : n.pred
5:          lock(p.succ_lock)
6:          s = s.succ
7:          if k ∈ (p.key, s.key] && !p.mark then
8:              if s.key ≥ k then
9:                  unlock(p.succ_lock)
10:                 return false
11:             parent = choose_parent(n,p,s)
12:             insert_to_tree(k,parent,s,p)
13:             unlock(p.succ_lock)
14:             grand_parent = lock_parent(parent)
15:             rebalance(grand_parent,parent);
16:             return true
17:         unlock(p.succ_lock)
```

Algorithm 6 Updates logical and physical layout fields after insertion

```
1:  function INSERT_TO_TREE(k, parent, s, p)
2:      new_node = create_node(k)
3:      new_node.parent = parent
4:      new_node.height = 0
5:      new_node.pred = p
6:      new_node.succ = s
7:      p.succ = new_node
8:      s.pred = new_node
9:      if k < parent.key then
10:         parent.left = new_node
11:     else
12:         parent.right = new_node
13:     parent.height= max(parent.left.height, parent.right.height) + 1
```

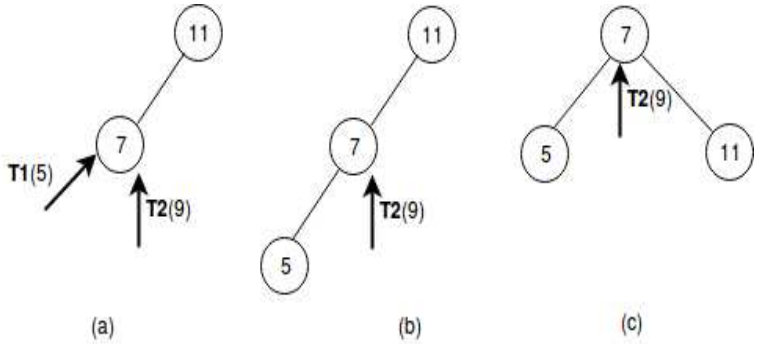


Figure 2.3: (a) Threads **T1** and **T2** try to acquire the tree_lock on N_7 (b) **T1** succeeds and inserts 5 in the tree (c) **T2** observes that N_7 can't be its parent anymore

2.3.3 Choosing Correct Parent

In sequential BST, the node returned by search operation is the actual parent, if its key is different than the key being inserted whereas the same may not hold in concurrent insertions. For example, in fig. 2.3(a) , two threads **T1** and **T2** try to perform *insert*(5) and *insert*(9) concurrently where *search*(5) and *search*(9) return the same node N_7. After determining the predecessor and successors using returned node, **T1** acquires *succ_lock* on $N_{-\infty}$ and **T2** acquires *succ_lock* on N_7. Next, **T1** acquires *tree_lock* on N_7(and hence **T2** gets suspended) and N_{11} and adds N_5 to the left of N_7 and observes that the balance factor at N_{11} is violated hence performs single right rotation to balance the tree and releases the acquired locks to be acquired by **T2** which observes that N_7 can't be its parent anymore and chooses N_{11} as its parent. It is guaranteed that either successor or predecessor would be chosen as parent once required tree_locks are acquired which would prevent any concurrent rotation at the locked node.

Algorithm 7 Choosing the correct parent

```
1:  function CHOOSE_PARENT(succ, pred, node)
2:      parent = (node==pred || node == succ ? node: pred)
3:      while true do
4:          lock(parent.tree_lock)
5:          if parent == pred then
6:              if parent.right == NULL then
7:                  return parent
8:              unlock(parent.tree_lock)
9:              parent = succ
10:         else
11:             if parent.left == NULL then
12:                 return parent
13:             unlock(parent.tree_lock)
14:             parent = pred
```

After choosing the correct parent, which is locked, physical and logical ordering layout fields are modified to bring about the changes after insertion as shown in algorithm 6. Next, attempt is made to lock the parent of the parent, i.e. grandparent of N_k and after acquiring the *tree_lock*, if grandparent node is marked, operation restarts otherwise it is chosen as the grandparent. Finally, rebalance operation is called on two locked nodes, i.e. parent and grandparent to update the heights of the nodes on the insertion path, if required, by performing appropriate rotations.

2.3.4 Delete Operation

Delete operation proceeds in the same way as *insert*. First, it calls search(k), determines the predecessor(p) of k in the tree using the returned node, acquires the *succ_lock* on p and applies the same validation as in insert. If validation succeeds, k is locked via *succ_lock* and required *tree_lock*s are acquired before marking the node to indicate that it was deleted from the tree. Next, logical ordering fields are updated by making N_p and N_k.succ to point to each other, i.e. N_p becomes predecessor of N_k.succ and N_k.succ becomes successor of N_p. Locks on N_p and N_k are released before updating physical layout fields of the tree. Physical deletion of the node k happens the same way as in sequential operation except affected nodes in concurrent operations are locked before performing updates. Heights are updated through rebalance operations which starts from original parent of N_k.succ if N_k had two children, N_k's parent if it had less than two children or N_k.succ if N_k was the parent of N_k.succ.

2.3.4.1 Acquiring Tree-locks

Removal of nodes having two children is carried out differently than those with less than two as different number of locks need to be acquired. The process starts by acquiring the *tree_lock* on the node(n) to be deleted followed by acquiring lock on its parent(p). On success, it tries to acquire *tree_lock*s on

- n's successor(s), parent of s and s's child(if it exist) when n has two children or
- n's child(if it exists) when n has less than two chlidren

If process fails to acquire lock on any of the nodes, it releases the acquired locks on other nodes and operations restarts from trying to acquire locks on n and its parent. It

Algorithm 8 Delete operation

```
1: function DELETE(k)
2:     while true do
3:         n= search(k)
4:         p= n.key < k ? n : n.pred
5:         lock(p.succ_lock)
6:         s= p.succ
7:         if k ∈(p.key, s.key] && !p.mark then
8:             if s.key > k then
9:                 unlock(p.itsucc_lock)
10:                return false
11:            lock(s.succ_lock)
12:            has_two_children =acquire_tree_locks(s)
13:            s.mark =true
14:            s_succ = s.succ
15:            s_succ.pred = p
16:            p.succ = s_succ
17:            unlock(p.succ_lock)
18:            unlock(s.succ_lock)
19:            remove_from_tree(s,has_two_children)
20:            return true
21:        unlock(p.succ_lock)
```

is necessary to check the number of children n has in each iteration as the number of children n has, may change due to concurrent updates from other threads. Operation terminates after successfully acquiring all the required locks. Physical removal of the node is performed the same as in sequential implementation.

2.3.5 Rebalancing

Update operations in AVL trees may create imbalance at certain nodes. One of four types of rotations are applied to balance the imbalanced nodes. Rebalancing process starts from the location of update and moves up till the root applying rotations, wherever required, to balance the tree. After insertion, it starts from the parent(p) and grandparent(g) of the inserted node; after deletion, it starts from the parent(g) of the deleted node and its child(p) if the node had less than two children(could be null if none) otherwise parent(g) and left-child(maybe null)(p) of the successor of the deleted node. During upward traversal of the tree, the rebalance operation locks *g* and *p* first to update their heights. Operation terminate if height of *g* does not change and height invariant property at *g* is not violated, it continues upwards by updating *p* to *g* and *g* to its parent if

Algorithm 9 Acquire Necessary tree_locks to delete the node

```
1: function ACQUIRE_TREE_LOCKS(n)
2:     while true do
3:         lock(n.tree_lock)
4:         r = n.right
5:         l = n.left
6:         if r == NULL || l==NULL then
7:             if r != NULL && !try_lock(r.tree_lock) then
8:                 unlock(node.tree_lock)
9:                 continue
10:            else if l != NULL && !try_lock(l.tree_lock) then
11:                unlock(node.tree_lock)
12:                continue
13:            return NULL
14:        successor= n.succ
15:        parent =n.parent
16:        if parent != n then
17:            if !try_lock(parent.tree_lock) then
18:                unlock(n.tree_lock);
19:                continue
20:            else if parent != successor.parent || parent.mark then
21:                unlock(parent.tree_lock);
22:                unlock(n.tree_lock)
23:                continue
24:        if !try_lock(successor.tree_lock) then
25:            unlock(n.tree_lock)
26:            if parent !=n then
27:                unlock(parent.tree_lock)
28:            continue
29:        succ_right = successor.right
30:        if succ_right != NULL && !try_lock(succ_right.tree_lock) then
31:            unlock(n.tree_lock)
32:            unlock(successor.tree_lock)
33:            if parent != n then
34:                unlock(parent.tree_lock)
35:            continue
36:        return successor
```

height of g changes and balance factor remains valid, rotation takes place otherwise. When balance factor at g is not valid and height of the subtree containing p is smaller than the other subtree, p is unlocked and attempt to acquire the lock on the other child of g is made. If attempt succeeds, p is made to point to the other child of g, otherwise all currently locked nodes are unlocked and attempt to acquire the locks on them restarts. If g is marked i.e. deleted, operation terminates.

CHAPTER 3

Overview of Cuda Architecture

Widespread demand for improving performance of consumer and industrial computing devices resulted in evolution of high-speed processors. However, due to various fundamental limitations in the fabrication of the integrated circuits, restrictions in heat and power consumption and limit in the size of transistors, it is no longer feasible to increase the CPU clock-speed in order to extract more powerful computation from existing architecture.

Graphics processing units(GPUs) were created to satisfy market demand for high speed realtime graphics processing. Since then it has evolved into highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth. Since GPUs are designed for highly parallel and compute intensive applications, more transistors are devoted to data processing rather than data caching and flow control.

3.1 Why GPU computing?

GPUs are designed to solve problems which show data-parallelism, i.e. same set of instructions execute on multiple data elements in parallel where operations on data involve more arithmetic computations than memory accesses. Since same set of instruction execute for each data element, control flow instructions are used quite less and as operations require intense arithmetic computations as compared to memory accesses, memory access latency can be hidden by greater arithmetic calculations instead of designing big and efficient data caches.

Generally CPUs contain just few cores which come with their own private and shared caches and process just 10s of software threads concurrently whereas a GPUs consist of hundreds of cores that can handle thousands of threads simultaneously. These capabilities enable GPUs to execute data-parallel task several times faster than CPU cores. Whereas GPUs provide execution speedup by diving task among thousands of threads, they are cost and power efficient as compared to CPUs.

3.2 Cuda

CUDA(*Compute Unified Device Architecture*) is a parallel computing platform which provides APIs to harness compute capability of NVIDIA GPUs. It provides libraries that enable data transfer between host and devices where host refers to CPU and its memory and device refers to GPUs and their memory. A typical sequence of operations during execution of cuda programs are:

- Declare and allocate memory on host and device
- Initialize host elements
- Copy host data to device memory
- Launch the kernels and perform computation
- Copy data back from device memory to host memory

CUDA organizes threads into hierarchy of *warps*, *blocks* and *grids*. It incorporates *global memory* concept which is shared by all the threads of the grid and *shared memory* which is shared by threads of a block. In order to ensure correctness of computation and avoid race conditions and deadlock, it provides *synchronization* primitives. The hierarchy in the thread organization helps programmers break their task into smaller and parallelizable tasks that can be executed independently by a warp/block of threads. CUDA runtime system manages execution of threads by launching thread blocks on available streaming multiprocessors(SM) and thereby relieving programmers from tedious task of thread management.

3.2.1 Programming Model

3.2.1.1 Kernels

CUDA kernels are the entry point for execution of programs on GPU cores. Kernels have the similar syntax as C functions but when invoked execute as many number of times as the number of threads launched and all concurrently. Unlike C functions, CUDA kernels have only *void* return type and their definition is preceded by *__global__* declaration specifier.

```
__global__ void kernel(param1, param2,...){
    statements;
}
```

CUDA kernels are invoked the same way as the functions in C except the kernel launch requires the number of threads to be specified that would execute it, which is specified within <<< .... >>>.

```
kernel<<<t_x,t_y,t_z>>>(param1,param2,...);
```

Depending on the values of kernel launch parameters within <<< ... >>>, a 3D/2D/1D grid of threads would be launched and each thread is assigned a unique id which can be used to control the execution of threads.

3.2.1.2 Thread Hierarchy

Cuda organizes threads into 1-dimensional, 2-dimensional or 3-dimensional blocks of threads. Blocks are organized into 1-dimensional, 2-dimensional or 3-dimensional grid of blocks as illustrated in Fig. 3.1. Threads within a block are indexed using **threadIdx** construct and blocks in a grid are indexed using **blockIdx**. Block and grid dimensions are accessed using **blockDim** and **gridDim** respectively. All the indexing and dimension constructs are three-field, namely x, y and z, structures. Local Id of a thread in a 3-dimensional block is calculated as:

```
local_threadId = blockDim.x * blockDim.y * threadIdx.
  z + blockDim.x * threadIdx.y + threadIdx.x
```

Similarly, block Id in a 3D grid is calculated as:

```
blockId = gridDim.x * gridDim.y * blockIdx.z +
  gridDim.x * blockIdx.y + blockIdx.x
```

and global Id of a thread is calculated as:

```
global_threadId = blockId * (blockDim.x * blockDim.y
  * blockDim.z) + local_threadId
```

Due to hardware limitations, the number of threads per block and blocks per dimension of the grid are restricted. Thread blocks execute independently. However, they can

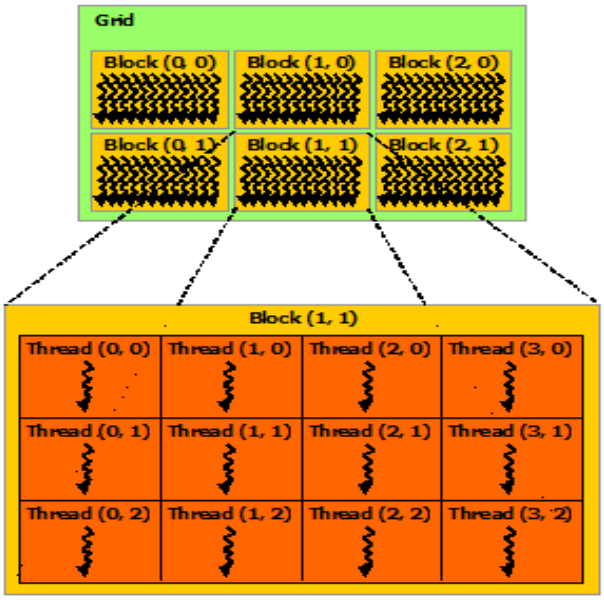


Figure 3.1: Organization of threads in grid and blocks

be made to execute in some specified order by using atomics and locks. Threads within a block can cooperate to perform some computations through shared memory. To coordinate memory accesses in such computations, threads are synchronized by calling *__syncthreads()* library function which acts as a barrier for the threads of each block.

3.3 Memory Hierarchy

Cuda organizes read and write memory into *global*, *shared* and *private* types. Global memory can be accessed by all the threads executing across all the cores. Shared memory is per black and can only be accessed by threads executing within the block. Local or private memory is thread specific and can only be modified by the thread it was allocated for as shown in fig. 3.2 Cuda also provides *constant* and *texture* memory spaces which are read-only and can be accessed by all the threads.

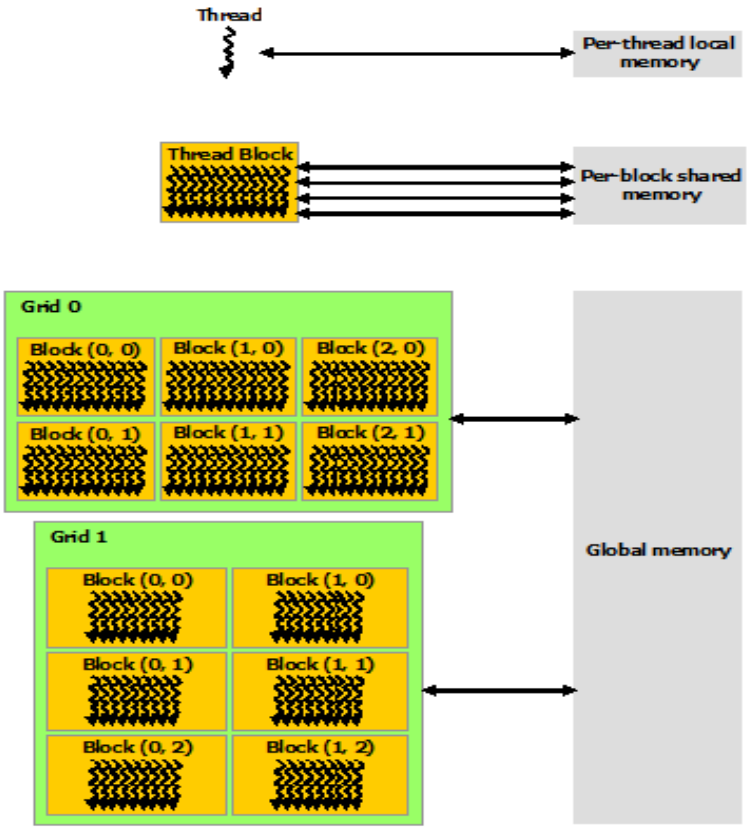


Figure 3.2: Device memory hierarchy

CHAPTER 4

Evaluation and Results

4.1 Dataset

Results were obtained by first inserting 1 million elements, created uniformly at random from the range (-10^8,10^8), in the tree. Before each test, test-data is generated uniformly at random from the range (-8N,8N) where N is the size of the dataset. Balanced Binary Search Trees are preferred over other data structures to store the data for their efficient search results. Therefore, in each test, number of search queries are in excess to the number of inserts and deletes. Test-data size ranges from 100 to 10 million and percentage of search, insert and delete queries in each dataset is kept as following:

- **type 1**: 100% search, 0% insert and 0% delete
- **type 2:** 50% search, 50% insert and 0% delete
- **type 3:** 50% search, 0% insert and 50% delete
- **type 4:** 50% search, 25% insert and 25% delete

4.2 Evaluation

Sequential and and concurrent cpu implementations were run on Intel i5-3210M processors with for processors and each consisting of 2 cores and operating at 2.50GHz. Concurrent cuda-implementation was run on NVIDIA's Tesla M2070 GPU cores. Performance is evaluated by comparing the times taken by all the threads to finish the execution of the operations assigned to them. Concurrent cpu implementation was tested by creating 128 threads for all sizes of the test data whereas concurrent gpu implementation was tested by creating as many number of threads as the number of queries when size of the test-data is less than 100000, 100000 otherwise. For each test-data size and mixture of query types, programs were run 10 times and time was taken as the average of all the execution. Following are the tabular and graphical representation of the results.

4.3 Observations

When all the queries are search type, GPU implementation performs > 1000 times faster than the other implementations for all sizes of the test data. GPU implementation is faster because each GPU thread has to execute < 100 queries as compared to ≈ 10000 queries executed by concurrent CPU threads.

Table 4.1: Test data consisting of 100% search, 0% insert and 0% delete queries

Data set size	**Sequential Execution time(ms)**	**Concurrent GPU time(ms)**	**Concurrent CPU time(ms)**
100	0.0257406	0.000066	2.6127337
1000	0.262193	0.000075	3.704370
10000	2.84989	0.000161	4.5367625
100000	34.435	0.001249	12.8423605
1000000	504.223	0.020151	54.135625
10000000	6760.34	0.285816	176.508964

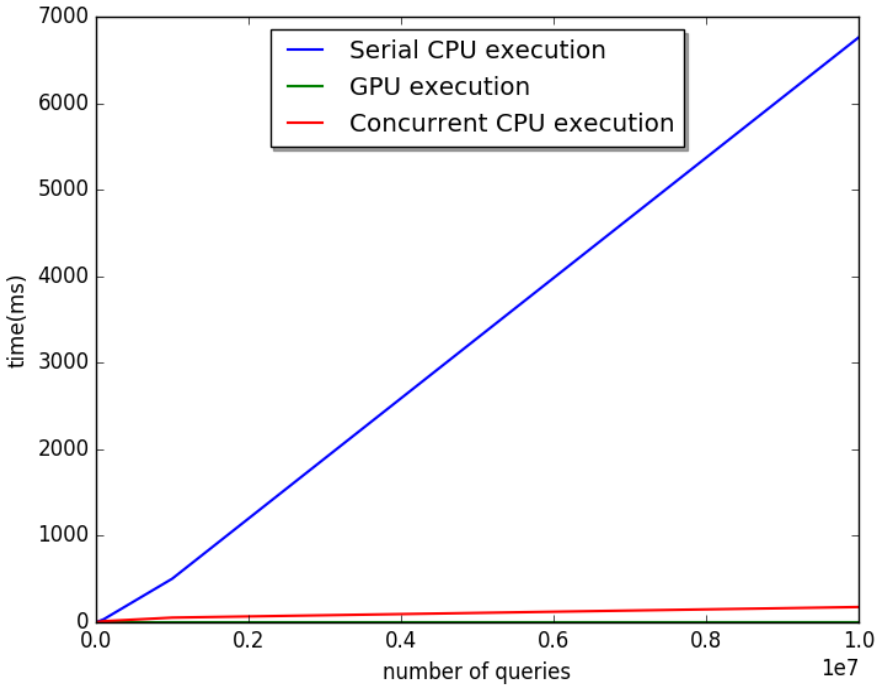


Figure 4.1: 100% search query. Integers used as keys

When mix of search, insert and delete operations are used, Concurrent CPU implementation catches up with GPU implementation owing to use of locks in for the

mutation operations

Data set size	Sequential Execution time(ms)	Concurrent GPU time(ms)	Concurrent CPU time(ms)
100	0.0604874	0.002042	3.4113046
1000	0.570185	0.021852	4.3562131
10000	6.02478	0.237240	8.2393948
100000	66.4861	2.442558	19.189123
1000000	874.748	25.450551	103.853393
10000000	11288.4	259.275844	>30 mins

Table 4.2: test data consisting of 50% search, 25% insert and 25% delete

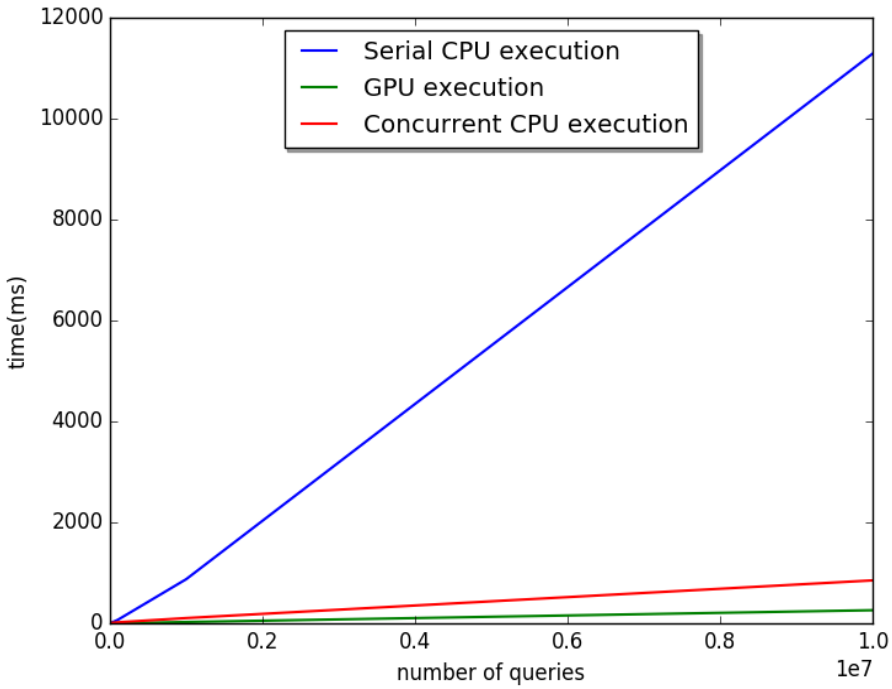


Figure 4.2: 50% search, 25% insert and 25% delete query 0%

When percentage of insert and delete operations is increased, CPU implementation catches up with the GPU implementation for large datasets, but fails to produce result within 30 mins for 10 million size dataset. Comparable performance of both the implementations can be attributed to the use of extensive locks in concurrent CPU implementation which makes the operations almost sequential as contention among threads increases with large number of queries resulting in threads taking up huge

amount of to time to acquire the locks, as compared to GPU implementation which performs searches(which constitute 50% of the queries) concurrently with inserts and deletes which are serialized to avoid the extensive use of locks which result in degraded performance.

Data set size	**Sequential Execution time(ms)**	**Concurrent GPU time(ms)**	**Concurrent CPU time(ms)**
100	0.0689402	0.002686	25.7077709
1000	0.677887	0.026318	32.625123
10000	7.35961	0.291068	47.826260
100000	77.9338	2.000197	59.2328736
1000000	1071.12	21.630555	972.607328
10000000	13826.7	232.478219	>30 mins

Table 4.3: test data consisting of 50% search, 50% insert and 0% delete queries

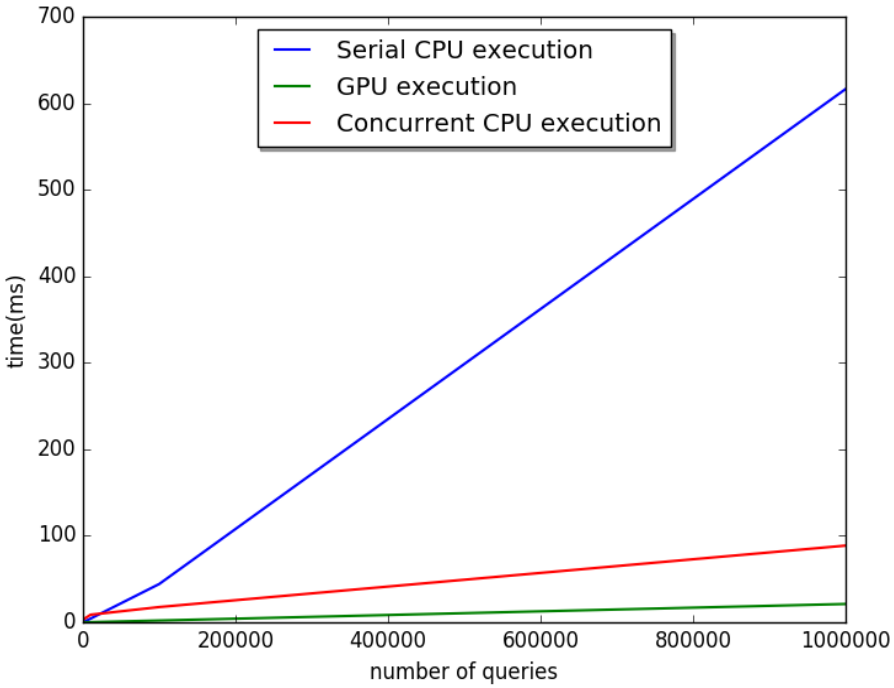


Figure 4.3: 50% search and 50% delete queries

Data set size	Sequential Execution time(ms)	Concurrent GPU time(ms)	Concurrent CPU time(ms)
100	0.0347251	0.000979	2.7682338
1000	0.36965	0.018600	3.65853025
10000	4.26217	0.197995	8.765159
100000	4.2681	2.054536	17.6363476
1000000	616.389	21.172125	88.64886775
10000000	7219.87	203.620203	>30mins

Table 4.4

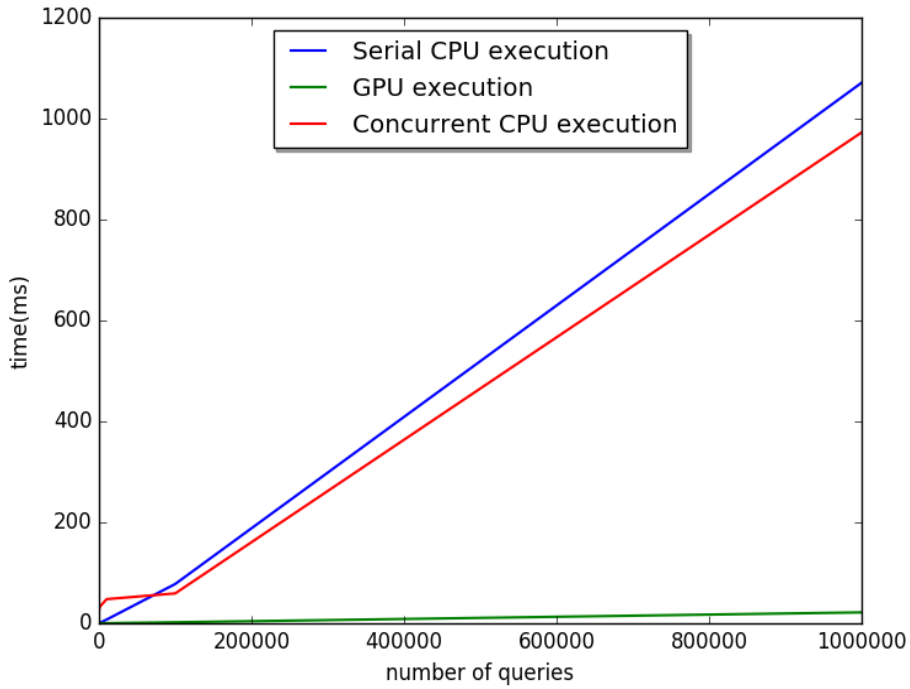


Figure 4.4: 50% search and 50% insert queries

CHAPTER 5

Future Work

Mutating operations are the bottleneck in the performance of algorithms on AVL trees. Locks are not advisable to be used on GPUs as they serialize the operations of the critical section they are used for. State of the art algorithms are designed to be run on CPUs but algorithms to to perform operations on AVL tree on GPUs have not been studied much. As parallel programming is becoming the requirement of almost all the applications for scalability and efficiency reasons, focus must be given on designing algorithms to perform lock-free operations on AVL tree as it is an excellent data structure to store the data and perform efficient search queries. Update operations, in AVL trees, require locking every node in the path while traversing up the tree, they become sequential when the number of queries increases. Balance factor constraint can be relaxed to mitigate the use of locks during concurrent operations. Logical Ordering of the tree elements allows concurrent lock-free search operations along with mutating operations. Similar approaches can be designed for insert and delete operations where certain number of insert or delete operations can be performed in some part of the tree like standard BST operations followed by rebalancing process for that part of the tree. Since much of the locking happen due to upward traversal of the operation upto root to rebalance the tree, constraints can be created to restrict the length of the path an operation would cover.

CHAPTER 6

Conclusion

The main focus of the implementation was to improve the performance of the search queries in concurrent environment on GPUs while performing other operations concurrently, which was achieved by implementing the Logical Ordering of the tree elements that is explicitly maintained in the tree and separates look-up operations from those that update the layout of the tree. Logical ordering enables efficient lock-free lookup operations along with other mutating operations in the tree. Extensive use of locks degrades performance and hence limits the amount of concurrency that can be extracted from AVL tree operations. For the aforementioned reasons and AVL tree being the height balanced binary search tree allowing $\log(n)$ search operations, it preferred over other trees in applications that require large number of search queries and very less number of insert and removals to be performed on the stored data. GPUs scale well when operations are lock-free but don't show much better performance when locks are used.

REFERENCES

[1] Mulralidhar Medidi and Narsingh Deo. *Parallel Dictionaries Using AVL Trees*. JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 49, 146-155 (1998) ARTICLE NO. PC981432

[2] Zhang Yin and Xu Zhuoqun. *Concurrent Manipulation of Expanded AVL Trees*[J]. Journal of Computer Science and Technology, 1998,V13(4): 325-336

[3] Joaquim Gabarró and Xavier Messeguer. *A Unified Approach to Concurrent and Parallel Algorithms on Balanced Data Structures*. Computer Science Society, 1997. Proceedings., XVII International Conference of the Chilean

[4] Joaquim Gabarró and Xavier Messeguer.*Parallel Dictionaries with Local Rules on AVL and Brother Trees*. ACM Journal on Information Processing Letters archive Volume 68 Issue 2, Oct. 30, 1998

[5] CARLA SCHLATTER ELLIS, *Concurrent Search and Insertion in AVL Trees*. IEEE Transactions on Computers, September 1980

[6] Nathan G. Bronson, Jared Casper. Hassan Chafi, Kunle Olukotun*A Practical Concurrent Binary Search Tree*. *PPoPP '10* Proceedings of the 15^th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming

[7] Dana Drachsler, Martin Vechev. Eran Yahav; *Practical Concurrent Binary Search Trees via Logical Ordering*. PPoPP '14 Proceedings of the 19^{th} ACM SIGPLAN symposium on Principles and practice of parallel programming

[8] Cuda C Programming Guide `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[9] Jason Sanders, Edward Kandrot;*Cuda By Example, An Introduction to General Purpose GPU Programming*

[10] David B. Kirk and Wen-mei W. Hwu; *Programming Massively Parallel Processors, A Hands-on Approach*