AN EFFICIENT IMPLEMENTATION OF NEAREST NEIGHBOUR MODELS ON MULTICORE GPU

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

MUKESH SHARMA

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2010

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled "**AN EFFICIENT IMPLEMENTATION OF NEAREST NEIGHBOUR MODELS ON MULTICORE GPU**" towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology** in **Information Technology** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from August 2009 to June 2010, under the guidance of **Dr. R. C. Joshi, Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

The matter presented in this dissertation has not been submitted by me for the award of any other degree of this or any other Institute.

Date: 18/06/2010

Place: Roorkee

[illegible]

(MUKESH SHARMA)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 18.6.10

Place: Roorkee

[illegible]

(Dr. R. C. JOSHI)

Professor

Department of Electronics and Computer Engineering

IIT Roorkee

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my heartfelt gratitude to my guide and mentor **Dr. R. C. Joshi**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his invaluable advices, guidance, encouragement and for sharing his broad knowledge. His wisdom, knowledge and commitment to the highest standards inspired and motivated me. He has been very generous in providing the necessary resources to carry out my research. He is an inspiring teacher, a great advisor, and most importantly a nice person.

Thanks are due to Dr. Padam Kumar, Head of Institute Computer Center, Indian Institute of Technology Roorkee, for providing workstation in Research Scholars' Lab.

I am greatly indebted to all my friends, who have graciously applied themselves to the task of helping me with ample moral supports and valuable suggestions.

On a personal note, I owe everything to the Almighty and my parents. The support which I enjoyed from my father, mother, elder sister, brother and brother in law provided me the mental support I needed.

MUKESH SHARMA

Abstract

The nearest neighbour problem is of practical significance in a number of fields such as bioinformatics, multimedia libraries, information retrieval etc. Finding an object near to a given query object is often a matter of interest. The problem is old, and a large number of solutions have been proposed in the literature. Many of these solutions need to build a model before answering the query. Most of the time the dataset is large or dynamic, we either opt for the strategy which does not need to build the model or move to some other solutions like approximations. The solution which uses some model may give answers efficiently but the drawback is model-building process, which is time consuming. These methods may answer the queries in less time but the model building process suppresses this advantage.

We have made an attempt to speed up the model building process using multicore architecture in order to reduce the pre-processing time. We have chosen cover tree data structure for model-building on multicore architectures because it scales well for large and high dimensional dataset.

In this dissertation entitled, "AN EFFICIENT IMPLEMENTATION OF NEAREST NEIGHBOUR MODELS ON MULTICORE GPU", a parallel algorithm for cover tree model building is proposed which implements a model of cover tree on Compute Unified Device Architecture (CUDA) in order to fasten the model-building process. In order to implement this parallel algorithm, an iterative cover tree algorithm is also proposed. A variant of cover tree data structure is also proposed for better parallelization. The proposed strategy can be extended to other algorithms which need model building.

We have implemented the proposed iterative cover tree algorithm for model-building using single core CPU and proposed cover tree algorithm for parallel architecture. We have also implemented our proposed variant of cover tree for single and multicore architectures. The results obtained from run time statistics of proposed algorithm show significant gain in speedup over linear algorithm.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction and Statement of the Problem

1.1 Introduction

Over the last decade, an immense amount of data has become available from collection of photos, genetic data, and network traffic statistics. Modern technologies and cheap storage devices have made it possible to accumulate huge datasets. The challenge is to effectively use this data. The ever growing size of the datasets makes it imperative to design new algorithms capable of sifting through this data with extreme efficiency. A fundamental computational primitive for dealing with massive dataset is the Nearest Neighbour (NN) problem. In the NN problem, the goal is to preprocess a set of objects, so that for a given query, one can efficiently find the data object most similar to that. A block B is said to be the neighbour of block A, if B has same property as A and covers an equal-sized neighborhood of A. NN problem has a broad set of applications in data processing and analysis. For instance, it forms the basis of widely used classification methods in machine learning that rely on nearest neighbour operations such as to give labels for new objects, dimensionality reduction algorithms, find the most similar labeled object and copy its label. Other applications include information retrieval, searching image databases, finding duplicate files and web pages, vector quantization, database queries, in particular for complex data such as multimedia, phylogenetic tree analysis [1] and in many others such applications.

Many solutions have been proposed for nearest neighbour problem. Some of them need to build model before answering the NN query and can answer query efficiently than others. The model building process suppresses this advantage which is time consuming. Hence there is a need to speed up the model-building process in order to take advantage of these methods for dynamic or large datasets.

1.2 Motivation

Many algorithms have been proposed in order to find the nearest neighbours in some given metric space such as naive approach, methods given in [2], [3], and [4], Navigating Nets [5], Cover Tree [6]. The Cover Tree data structure is relatively new and introduced first by Beygelzimer et al. [6] and was proved to be efficient in terms of space complexity as well as time complexity for constructing the tree (pre-processing) and nearest neighbour search as compared to other data structures and methods.

The cover tree model building is a pre-processing stage to find the nearest neighbours. Since in most of the applications, the dataset is very large or dynamic and the tree construction is time consuming, speedup in tree construction is required to find the closest points fast. Serialization has a limit and cannot be improved further but parallelization is a way to make computationally intensive algorithms more efficient.

Recently, Graphics Processing Units (GPUs) feature hardware was optimized for simultaneously performing many independent floating-point arithmetic operations in the display of 3D models and other graphic tasks. Thus, general-purpose GPU (GPGPU) programming has been successful primarily in the scientific computing disciplines which involve a high level of numeric computation. However, other applications can also be successful, provided the applications feature significant parallelism. As the GPU has become increasingly more powerful and ubiquitous, researchers have begun exploring ways to tap its power for non-graphics, or GPGPU applications [7].

Multicore architecture seems to be a good option for porting such applications. When comparison is made on basis of cost, multicore systems are cheaper than multiprocessing systems or even grids. They consume less power and can be used for a wide variety of applications. CUDA devices can be used as add on in most of existing systems and can perform various task. Hence use of such devices to speed up the model building pre-processing stage will be highly beneficial.

1.3 Statement of the Problem

The problem is to design a model which can speed up the model-building process in the pre-processing stage of the nearest neighbour search algorithms using multicore architectures such as CUDA GPU.

We have made an attempt to handle the aforementioned problem in the following ways:

- Designing and implementing algorithm for cover tree model-building using CUDA GPU.
- To explore the possibilities of variants which overcome the limitations of cover tree for implementation on multicore architectures and yields better parallelization.

1.4 Organization of the Report

This dissertation report comprises of six chapters including this chapter that introduces the topic and states the problem. The rest of the report is organized as follows.

Chapter 2 gives the background information about nearest neighbour problem and discusses some well known algorithms for nearest neighbour search.

Chapter 3 gives the introduction of cover tree and a brief explanation of algorithms for various operations on cover tree. We also discuss the hardware architecture of CUDA.

Chapter 4 describes parallel implementation of Cover Tree batch insertion algorithm on CUDA architecture. We also propose a variation of cover tree and its implementation on single core and CUDA architecture.

Chapter 5 discusses the runtime statistics of linear and parallel versions of cover tree implementation algorithm, and parallel version of implementation of proposed variant of cover tree.

Chapter 6 concludes the dissertation work and gives suggestions for future work.

Chapter 2

Background and Literature Review

2.1 Nearest Neighbour Problem : Definition

Finding the nearest neighbour of a point in a given metric is a classical algorithmic problem. The basic nearest neighbour problem can be formally defined as:
Given a set S of n points in some metric space (X, d), the problem is to pre-process S so that given a query point $p \in X$, one can efficiently find a point $q \in S$ which minimizes $d(p, q)$.

In other words we can say, in some vector space endowed with a distance function (typically a d-dimensional Euclidean space), we are given a set of n points (called the database). Given any other point (called a query), we must find the closest point to it in the database. We have to pre-process the database efficiently and create a data structure that will support efficient search. More specifically, the trivial data structure storing the unprocessed list of points allows us to search spending *O(nd)* arithmetic operations.

2.2 Extensions and Variations of the Problem

A natural and straight forward extension of this problem is k-nearest neighbour (kNN) search, in which we are interested in the k ($\leq |S|$) nearest points to q, contained in S. The NN search then just becomes a special case of kNN search with k=1. A slight variation of NN search, advocated by Indyk et al. [8], and Datar et al. [9] in place of NN search, is ε -approximate NN (ε-NN) search, where given a user defined error bound $\varepsilon > 0$, the task is to find a point p' in S which is at most ε times farther than the exact NN (also in S), i.e. $p, p' \in | S \leq M(p', q) \leq (1 + \varepsilon)M(p, q)|$. For the kNN case this simply extends to finding k neighbours $p_1', p_2', \ldots p_k'$ such that any of the ith ε-approximate NN p_i' is at most ε times farther than the exact ith NN p_i, i.e. $M(p_i', q) \leq (1 + \varepsilon)M(p_i, q)$. Figure 2.1 illustrates this graphically.

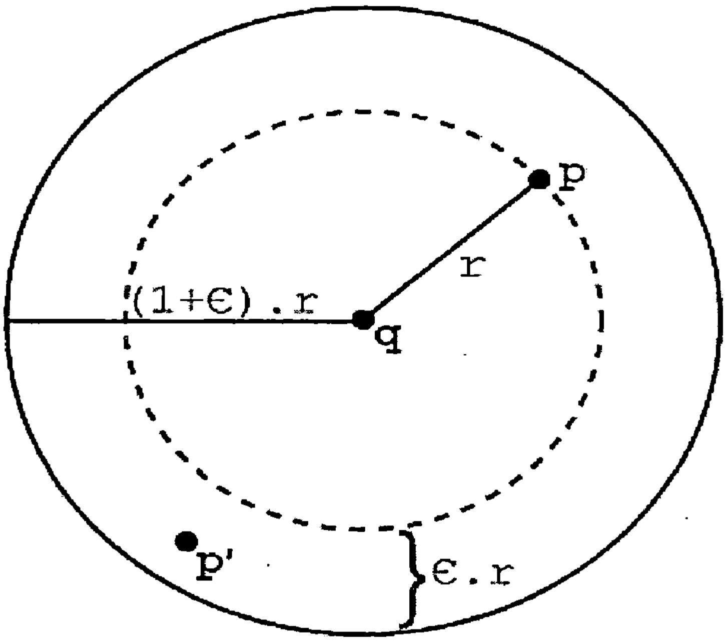

Figure 2.1 ε-Approximated Nearest Neighbour

2.3 Applications of the NN Problem

The kNN search is of practical significance in a number of fields. Some of those, along with examples of their use of kNN search, include:

- **Data compression:** Here, it is used in a method called vector quantization for speech and image compression as given by Gersho et al. [10]. It involves blocking speech or image waveform signals into vectors of fixed length. A set of codevectors is first computed based on a set of training vectors, and then each new vector is encoded with the index of its nearest neighbour among the codevectors.
- **Pattern recognition, data-mining and machine learning:** Here, one of the most widely used classifier/learner is the kNN classifier as shown by Cover et al. [11]. It is based on straight forward adoption of the kNN search, and works by assigning a given test point the majority class of its k-nearest neighbours. Also as given by Atkeson et al. [12], locally weighted learning is another technique which utilizes kNN search. It works by training its base classifier/learner on training points that are nearest neighbours of a given test point.

- **Bioinformatics:** Here, kNN and its variant classifiers have been applied successfully to biological and clinical data. They have been used for cancer classification as described by Niijima et al. [13], for detecting rRNA sequences [14], and, using gene-expression data for tumour classification [15] and tissue classification [16]. In cases involving gene selection [13], [15], [16], these classifiers have been observed to perform as well or even better than state-of-the-art SVM based classifiers.
- **Multimedia databases and libraries:** Here, similarity search is often used to retrieve multimedia content similar to a user query. The systems usually allow content-based queries, i.e. queries in the form of object shapes, texture, dominant colours, and scene descriptions etc. for images and video as described in [17], and in the form of dominant frequency and pitch (which can also be given as an acoustic input from the user) etc. in case of an audio/music library [18].
- **Computer vision:** Here, NN search is an important tool used for the task of object classification, which involves finding similarities between images. It has also been applied recently in a prototype of a robust multi-target tracking system, which tracks players in a hockey rink [19].
- **Document/information retrieval:** Here, NN search methods are often used to retrieve and rank documents given a user query [20], [21].

2.4 Literature Review

A brute force approach to find a nearest neighbour would take O(n) time to evaluate each of n points in a set. Since there might be millions of points, and assessment might be expensive, time can become a serious issue. Consider also that a 128-dimensional point, using 8 bit integers occupies 128 bytes. 1 million points would take 125 MB. 32 million points would take 4 GB. At this number, available memory could become a serious issue as well. Nearest neighbour search algorithms can be assessed on their use of memory, storage, and processing power.

2.4.1 k-Dimensional Trees (KDTrees), BBF-Trees and Variants

Multidimensional binary search trees, called in short by the author as KDTrees (where *k* is the dimensionality of the space), were originally proposed by Bentley [22] for associative retrieval of records in a file. Their potential for NN search was observed by Bentley, and hence were quickly adopted for NN searching, with an optimized version by Friedman, Bentley and Finkel [23]. Since then, KD Trees are by far the most popular search technique employed for NN search.

The trees hierarchically partition the point space into mutually exclusive rectangular regions by recursively splitting it with axis-parallel hyper planes. The splitting is binary, with each non-terminal internal node splitting a region into two sub regions. The search for (k)NN is carried out recursively, with the region containing the query point being searched first and then only those of the remaining ones which are likely to contain the (k^{th})NN. More specifically, after recursively narrowing down to the region of a leaf node containing the query, the points inside the region are looked at, and then a ball (hypersphere to be exact), centred at the query and with radius equal to the query's distance to the best (k^{th})NN found so far, is computed. Afterwards during backtracking only those regions which intersect with this query ball are searched, and the ball is updated each time a better (k^{th}) NN is encountered in another region.

The trees require data in vector representation. They utilize this representation very efficiently and do not require any distance computation either during construction or during much of NN search (distance computations are performed only when looking at points inside a region of a leaf during the NN search). During both processes they only look at the value of a point's dimension that is orthogonal to the hyperplane used to split a region.

The original version proposed requires O(dnlogn) construction (pre-processing) time, and O(n) storage. For a given query it takes O(logn) time in the expected case for moderate dimensions. The query time, however, usually grows exponentially in d, and the tree, suffering from the curse-of-dimensionality, usually degenerates to simple linear search at higher dimensions (with slightly higher query time). The original version is general enough to be applied even to non-metric distance measures, and requires measures to satisfy only a few constraints given by the authors. Still, however, it is only known to be evaluated and found efficient. Also in KD trees, the node divisions are always axis-aligned, regardless of the data distribution. This often results in poor search performance.

In real applications, the first problem is typically skirted by relaxing the requirement that all close neighbours be found. The Best Bin First (BBF) approach [24] is one such technique. It is based on the observation that the vast majority of the neighbouring cells usually do not contain a nearest neighbour. It therefore searches the candidate cells in ascending order of their distance to the query, and terminates the search early to save computations. In [25], it is claimed that the method produces 95% of the correct neighbours at 1% the cost of an exhaustive search for one particular application. However, we note that this does not guarantee that all nearest neighbours are found, which can cause problems for various applications (e.g., as described in [26]).
The drawbacks can be summarized as:
1. The number of neighbours for each leaf node grows exponentially with dimension, causing search to quickly devolve into a linear scan.
2. The node divisions are always axis-aligned, regardless of the data distribution. This often results in poor search performance.

2.4.2 Ball Trees

The kd-tree and its variants can be termed "projective trees," meaning that they categorize points based on their projection into some lower-dimensional space. In contrast, all our remaining methods are "metric trees" – structures that organize points based on some metric defined on pairs of points. Thus, they don't require points to be finite-dimensional or even in a vector space.

In their original form, each node's points are assigned to the closest centre of the node's two children. The children are chosen to have maximum distance between them, typically using the following construction at each level of the tree. First, the centroid of the points is located, and the point with the greatest distance from this centroid is chosen as the centre of the first child. Then, the second child's centre is chosen to be the point farthest from the first one. The resulting division of points can be understood as finding the hyperplane that bisects the line connecting the two centres, and perpendicular to it. Note that in this construction, there is no constraint on the number of points assigned to either node and the resulting trees can be highly unbalanced. While unbalanced trees are larger (and take longer to construct) than their balanced counterparts, this does not mean that they will be slower to search. On the contrary, such trees might be significantly faster if they capture the true distribution of points in their native space.

However the ball trees are good if range search is needed but following restrictions can be seen with ball tree data structure:

- Tree construction algorithm does not scale to very large datasets.
- A ball in R^D is not such a useful region shape if sample density varies in the feature space.

2.4.3 k-Means

While the previous description of ball trees is probably familiar to members of the machine learning community, we can notice its similarity to the k-means method [27]. This algorithm also assigns points to the closest of k centres, although it does so by iteratively alternating between selecting centres and assigning points to the centres until neither the centres nor the point partitions change. As originally described, the k-means method is a simple non-hierarchical clustering method that requires careful selection of both k and the initial centres to avoid local minima and bad partitions. Linde et al. [28] extend this method to a hierarchical structure where k now defines the branching factor between successive levels of the tree.

Following disadvantages have been observed when using this technique:

- Difficulty in comparing quality of the clusters produced (e.g. for different initial partitions or values of K affect outcome).
- Fixed number of clusters can make it difficult to predict what K should be.

- Does not work well with non-globular clusters.
- Different initial partitions can result in different final clusters. It is helpful to rerun the program using the same as well as different K values, to compare the results achieved.

2.4.4 Navigating Nets and Cover Trees

These structures try to exploit the intrinsic dimensionality of a dataset (i.e. data points plus the query points). They work by placing assumptions that the datasets (or the metric spaces in which they are embedded), regardless of their actual number of dimensions, exhibit certain restricted or bounded growth. A simple notion of such bounded growth was presented by Karger and Ruhl [4]. They defined a growth bound on a dataset such that the number of points in a ball (hypershere to be precise) centred at any point p is at most c times the number of points in a ball of half the radius centred at the same point; more formally, for all points p (in the dataset) and for all radii $r > 0$, $|B(p, 2r)| \leq c \leq |B(p, r)|$. Their presented growth bound only allows points to come into view at a constant rate c (called the expansion rate), and rules out the possibility of suddenly encountering an exponentially high number of points as the ball around p is expanded. Such growth, as pointed out by Karger and Ruhl, occurs naturally in domains like peer-to-peer networks and the Internet. Karger and Ruhl also presented a data structure for NN search which works well for geometries/datasets satisfying their growth bound.

A similar bound property was defined by Krauthgamer and Lee [5]. Their growth bound, however is more general than the one by Karger and Ruhl. Their growth bound definition is, "every set of points in the dataset should be able to be completely covered with at most 2^ρ sets of half the diameter" or in other words, "any ball about a point p in the dataset should be able to be completely covered (in terms of the points it contains) with at most 2^ρ balls of half the radius." They defined the intrinsic or abstract dimensionality of a dataset, using this growth bound, as the minimum ρ for which this bound holds. This growth bound forms the basis of Navigating Nets, data structures for ε-NN search, which the authors also presented with their growth bound in [5].

Navigating Nets work by arranging the points in levels, such that each lower level acts as a cover for the previous level, and each lower level has balls half the radius than the ones at the previous level. The top level consists of a single point with a ball centred at it that has radius $2^{i'}$ for an i' big enough to cover the entire set of data points. The next level consists of points with balls of half the radius than the top most ball ($2^{i'}$-1), which cover the points at a finer level. The bottom-most level consists of points that have balls covering only those single points. The structure is built in a greedy manner, where the first point in the list of points of the top level ball is used to build a smaller ball at the next level, and the first point inside the smaller ball is used to build a ball smaller at the level next. This is done recursively until we reach a level where a ball consists of a single point (on which that ball is centred). Then the build procedure back tracks to the last higher level cover ball that still has unprocessed points left, and picks the next point to greedily build cover balls at lower levels. Using the same terminology as Krauthgamer and Lee, if dmax is the maximum of the inter-point distances of the data points, dmin the minimum, and Δ = dmax/dmin the ratio between the maximum and the minimum inter-point distances, then the number of levels of a Navigating Net on a dataset is O(logΔ). If the dataset satisfies the growth bound given by authors, then every ball has at most O(1) cover balls at the lower level (because of the constant Δ). The search for a (k)NN of a given query q is carried out by going down the levels of a Navigating Net and adding to a set of candidate NNs the children of a point whose ball intersects with the ball centred at the query. The radius of the query ball is set to the distance of the current best (kth)NN plus the radius of cover balls at the level currently being looked at. The search begins by adding the top-most point to the set of candidate NNs and setting the radius of the query ball as mentioned, so as to cover the entire point space, and then adding all the children of the top point to the set of candidate NNs. Then the search descends to the lower level, contracts the radius of the query ball (to the distance of the current best NN plus the radius of cover balls at this lower level) and adds the children of only those children of the top point whose balls intersect with the contracted query ball. The search carries on in this manner until we reach the bottommost level or if at some level the current best (kth)NN cannot be at a distance more than ε farther from the query than the exact (kth)NN. Hence, at the end of the search the current best (k)NN is the ε-NN of the given query.

For a dataset satisfying the authors' growth bound, the search procedure takes no longer than O(logΔ), and if the dataset instead satisfies the growth bound of Karger and Ruhl (which is a special case of Krauthgamer and Lee's growth bound), then the search procedure takes no longer than O(logn). Beygelzimer et al. [6] presented a data structure for NN and ε-NN search based on Navigating Nets, which they called Cover Trees. In a Navigating Net each point at some lower level is allowed to have more than one parent point from the previous level (i.e. points are allowed to overlap among balls in any intermediate level), and also each level has also a pointer to the previous top level. In Cover Trees, the authors removed these redundancies to convert the graph rendered by a Navigating Net into a tree, while still preserving the construction and query time. Furthermore, they also used the growth bound of Karger and Ruhl, as the one by Krauthgamer and Lee does not have strong theoretical guarantees for exact NN search.

Navigating nets perform well but it does not guarantee to find exact nearest neighbour and are used for approximate NN search. Cover tree scales well for large dataset and answers query efficiently but in all tree structures a pre-processing is needed which consumes time and hence cannot be used for dynamic datasets.

Chapter 3
Cover Tree Data Structure and CUDA Architecture

3.1 Cover Tree

3.1.1 Introduction

Beygelzimer et al. [6] introduced cover tree data structure for fast nearest neighbour operations in general n point metric space. Also, cover tree was proved to be efficient in terms of space complexity as well as time complexity for constructing the tree (preprocessing) and nearest neighbour search as compared to other data structures and methods.

Cover tree is relatively a new data structure. Independent of the doubling dimension, the space used is O(n). Further, nearest neighbour queries can be done in $O(c^{12}\ln(n))$ time with insertion and removal taking $O(c^{6}\ln(n))$. An example of a cover tree can be seen in Figure 3.1. Note that each point is compared using the Euclidean metric. At first glance, it might appear that once a node appears it is in the tree forever, that at higher levels nodes seem to be relatively far apart and that children are somewhat close to the parents.

A cover tree T on a dataset S is a leveled tree. Each level is a cover for the level beneath it, i.e. level i contain all the elements containing in level i+1. Each level is indexed by an integer scale i which decreases as the tree is descended. Every node in the tree is associated with a point in S. Each point in S may be associated with multiple nodes in the tree; however, it is required that any point appears at most once in every level. A node in the tree may have many children node but it can have only one parent node.

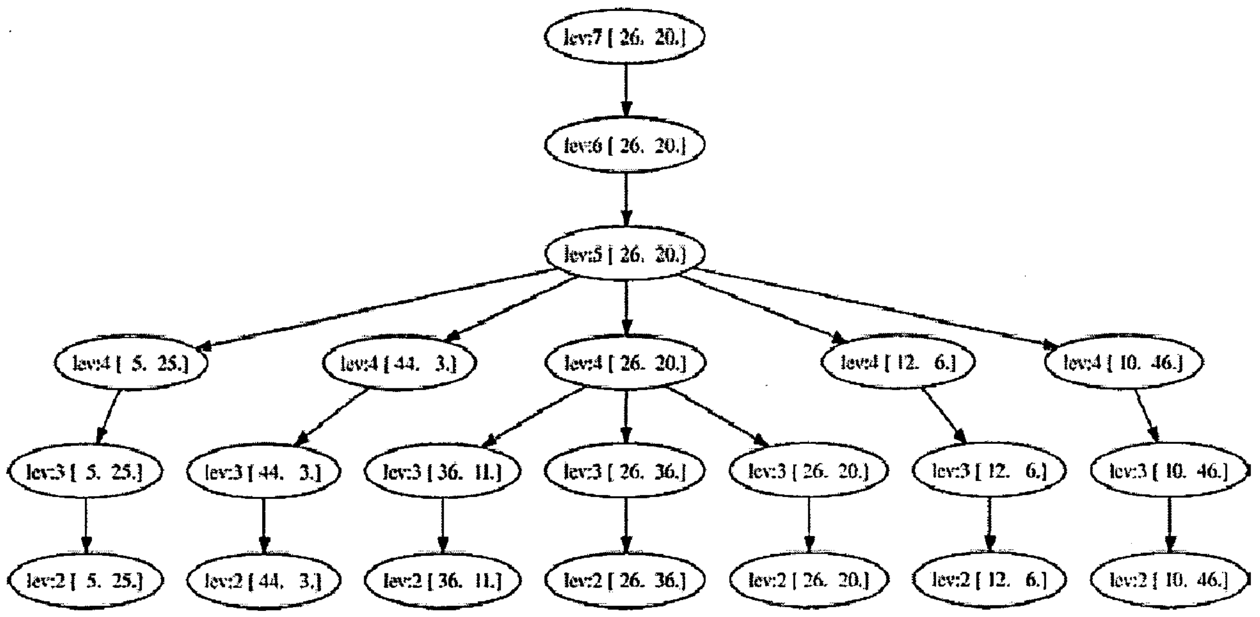

Figure 3.1 Cover Tree data structure [25]

Let C_i denote the set of points in S associated with the nodes at level i. The cover tree obeys the following invariants for all i:

(1) $C_i \subset C_{i-1}$ (nesting). This implies that once a point $p \in S$ appears in C_i then every lower level in the tree has a node associated with p.

(2) $\forall\, p \in C_{i-1}$, there exists a $q \in C_i$ such that $d(p, q) \leq 2^i$ and the node in level i associated with q is a parent of the node in level $i - 1$ associated with p (covering tree).

(3) $\forall\, p, q \in C_i$, $d(p, q) > 2^i$ (separation) i.e. at each level, distance between any two points is greater than 2^i.

Let us take a closer look at these three properties. In particular, the nesting property states if a point p is in the tree at the ith stage, then it is also in the tree at the i−1st stage. If we view this on a scale from ∞ to $-\infty$, then intuitively there should be only one point at ∞ and the dataset S at $-\infty$. See figure 3.2

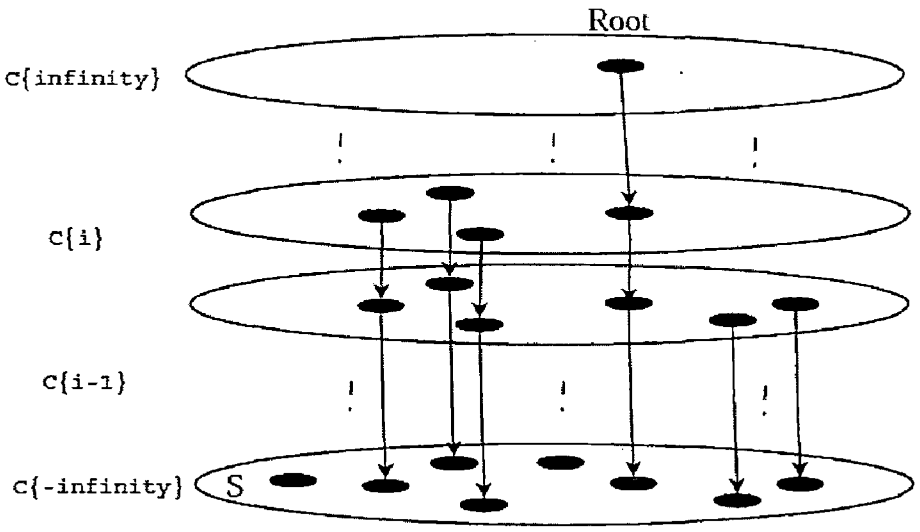

Figure 3.2 Graphical representation of Nesting Property [29]

The covering tree property says that every node at the i−1st stage has exactly one parent that is within distance 2^i of it. In other words, there exists only one path to each node in tree, with the parents being close to the children (close with respect to the metric d). See figure 3.3

Finally, the separation property states that every node at the ith stage is separated by at least 2^i. This means that at every level in the tree, the nodes are far away from one another with respect to the metric d. See figure 3.4

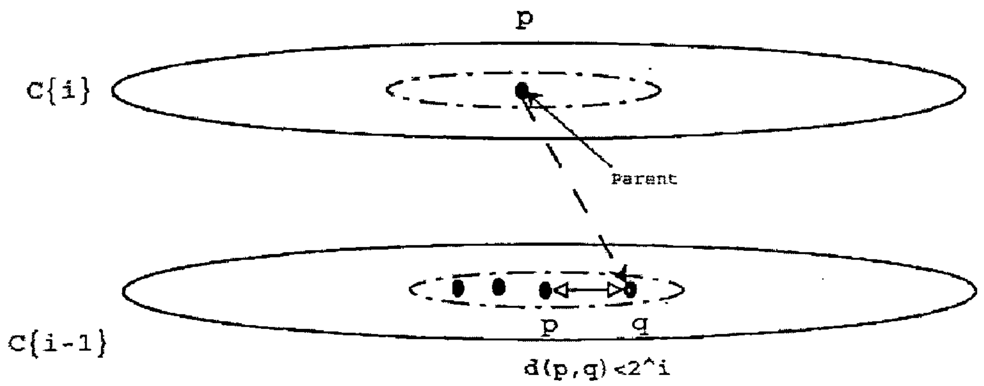

Figure 3.3 Graphical representation of covering tree Property [29]

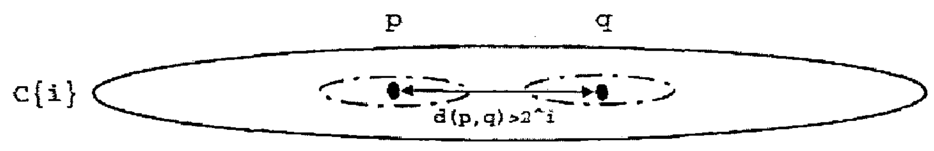

Figure 3.4 Graphical representation of separation Property [29]

3.1.2 Algorithms for Various Operations on Cover Tree

The insertion algorithm given by Beygelzimer et al. [6], is shown in algorithm 1 but Kollar [25] proved that the algorithm was having minor bugs and gave the correct algorithm which is as given in algorithm 2. In the faulty algorithm given in [6], if "no parent found" is interpreted to be false and "parent found" to be true, then it can be seen that this algorithm will fail to maintain the cover tree properties. In particular, it is easy to see that the covering property will fail to hold. Say that the recursion bottoms out at the first if statement and that the recursion depth is n. Then, the algorithm recurse up to depth n−1 and run the else statement, which will go into the second if statement. Now, this will return true and add a parent of p. Popping up another level of recursion to n−2, the last else statement is entered and false is returned. One more recursion to n−3 causes the algorithm to perhaps enter the second if statement again. Thus, the algorithm has added two elements to be parents of the point p, which violates the covering property of cover trees.

Algorithm 1 Original faulty Insert procedure for Cover Trees

```
(1)   Insert(point p, cover set $Q_i$, level i):
(2)   $Q = \{Children(q) : q \in Q_i\}$
(3)   if $d(p,Q) > 2^i$:
(4)       return "no parent found"
(5)   else:
(6)       $Q_{i-1} = \{q \subset Q : d(p, q) \leq 2^i\}$
(7)       if Insert(p, $Q_{i-1}$, i − 1)=="parent not found" and $d(p,Q_i) \leq 2^i$
(8)           pick a single $q \in Q_i$ such that $d(p, q) \leq 2^i$
(9)           insert p into Children(q)
(10)          return "parent found"
(11)      else:
(12)          return "no parent found"
```

Algorithm 2 Insert procedure for Cover Trees

```
(1)  Insert(point p, cover set $Q_i$, level i):
(2)  $Q = \{Children(q) : q \in Q_i\}$
(3)  if $d(p,Q) > 2^i$:
(4)      return “parent found” - True
(5)  else:
(6)      $Q_{i-1} = \{q \in Q : d(p, q) \leq 2^i\}$
(7)      found = Insert(p, $Q_{i-1}$, $i - 1$)
(8)      if found and $d(p,Q_i) \leq 2^i$
(9)          pick a single $q \in Q_i$ such that $d(p, q) \leq 2^i$
(10)         insert p into Children(q)
(11)         return “finished” – False
(12)     else:
(13)         return “found”
```

The algorithm for nearest neighbour search through cover trees is given in algorithm 3. To find the nearest neighbour of a point p in a cover tree, the algorithm descends through the tree level by level, keeping track of a subset $Q_i \in C_i$ of nodes that may contain the nearest neighbour of p as a descendant. The algorithm iteratively constructs Q_{i-1} by expanding Q_i to its children in C_{i-1} then throwing away any child q that cannot lead to the nearest neighbour of p. For simplicity, it is easier to think of the tree as having an infinite number of levels (with C_∞ containing only the root, and with $C_{-\infty} = S$). Denote the set of children of node p by Children(p) and let $d(p,Q) = \min_{q \in Q} d(p, q)$ be the distance to the nearest point of p in a set Q. Note that although the algorithm is stated using an infinite loop over the implicit representation, it only needs to operate on the explicit representation.

Algorithm 3 Find-Nearest (cover tree T, query point p)

```
(1) Set $Q_\infty = C_\infty$, where $C_\infty$ is the root level of T.
(2) for i from $\infty$ down to $-\infty$
(3)     Set $Q = \{Children(q) : q \in Q_i\}$
(4)     Form cover set $Q_{i-1} = \{q \in Q: d(p, q) \leq d(p,Q) + 2^i\}$
(5) return arg min $q \in Q_{-\infty}$ d(p, q).
```

Algorithm 4 Remove(point p, cover sets $\{Q_i, Q_{i+1}, ..., Q_\infty\}$, level i)

```
(1)  set $Q = \{Children(q) : q \in Q_i\}$
(2)  set $Q_{i-1} = \{q \in Q : d(p, q) \leq 2^i\}$
(3)  Remove(p, $\{Q_{i-1}, Q_i, ..., Q_\infty\}$, i − 1)
(4)  if d(p,Q) = 0 then
(5)      remove p from $C_{i-1}$ and from Children(Parent(p))
(6)      for every $q \in$ Children(p)
(7)      set i' = i − 1
(8)      while $d(q, Q_{i'}) > 2^{i'}$
(9)          insert q into $C_{i'}$ (and $Q_{i'}$) and increment i'
(10)         choose $q' \in Q_{i'}$ satisfying $d(q, q') \leq 2^{i'}$ and make q' point to q
```

3.1.3 Explicit and Implicit Cover Trees

The cover tree can be constructed in two ways: implicit representation and explicit representation. In implicit representation once the point is inserted at some level in tree then it will be inserted in all levels until all the points are inserted in the tree while in explicit representation, once the point is the parent of only itself then it is removed from all of the levels below it. The explicit and implicit representations are shown in figure 3.5 and figure 3.6 respectively. Theory is based on an implicit implementation, but tree is built with a condensed explicit implementation to preserve O(n) space bound.

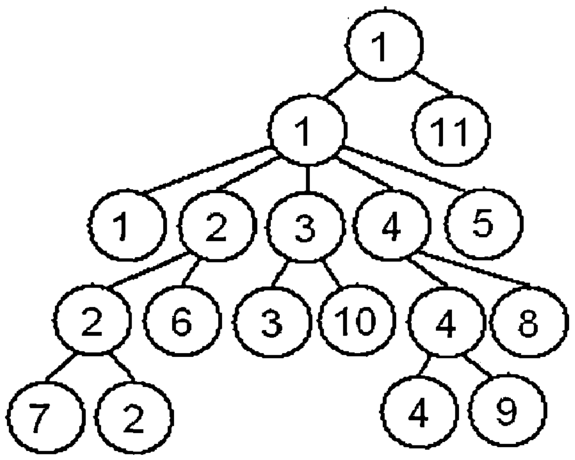
Figure 3.5 Explicit representation of cover tree

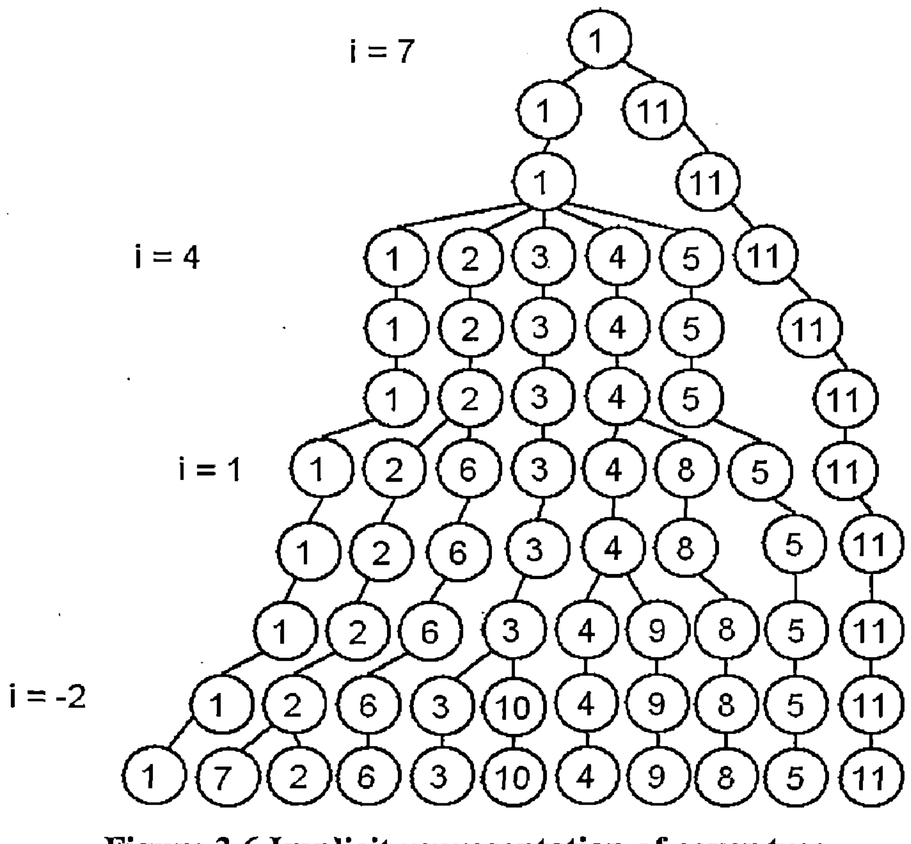

Figure 3.6 Implicit representation of cover tree

The cover tree structure can also be used to approximate nearest neighbours. Given a point p ∈ X and some **ε** > 0, we want to find a point q ∈ S satisfying d(p, q) < (1+ **ε**) * d(p, S). The main idea is to maintain a lower bound as well as an upper bound, stopping when the interval implied by the bounds is sufficiently small.

3.2 nVidia CUDA architecture

We now discuss a little about nVidia's CUDA (Compute Unified Device Architecture) architecture which we have used to parallelize Cover Tree algorithms. General purpose computing on the GPU is an active area of research. GPUs are already widespread. The performance of GPUs is improving at a rate faster than that of CPUs. The capabilities of the GPU have increased dramatically in the past few years and the current generation of GPUs has higher floating point performance than the most powerful (multicore) CPUs [30]. The GPU contains hundreds of cores that work great for parallel implementation. The programming is done in SIMD style where same code is worked on different data locations. Until recently a graphics API was needed to code on GPUs which made coding for non graphics oriented calculations tough. Trying to work around this limitation nVidia released CUDA which allows GPUs to be programmed using a variation of C. This enables a low learning curve and makes programming easier.

The three abstractions of the CUDA model are a hierarchy of thread groups, shared-memories, and barrier synchronization. Threads are arranged in the form of a grid which is a two dimensional array of thread blocks. Each thread block is a three dimensional structure that houses the threads. This type of hierarchy is given to the programmer so that the arrangement of the threads is similar to the way programmer's data is arranged (in arrays). Threads within a block can cooperate among themselves by sharing memory. Shared memory is expected to behave like an L1 cache where it resides very close to the processor core. Synchronization points can be specified by calling the function __syncthreads.

The memory available to the threads is of three types. Every thread has local memory. Number of threads which are in the same thread block can share memory. And the third type of memory is the global memory that every thread has access to. C code for both the GPU and the CPU resides in the same file. The CPU code follows a sequential flow. GPU code is called by a kernel call. This is where the code runs in parallel. A large number of threads are created by the kernel call. These threads then run parallely on the GPU.

Figure 3.7 shows the tremendous computational capability of the GPU. GTX 280 a GT 200 family GPU delivers a peak performance of 933 GFLOPS/sec.

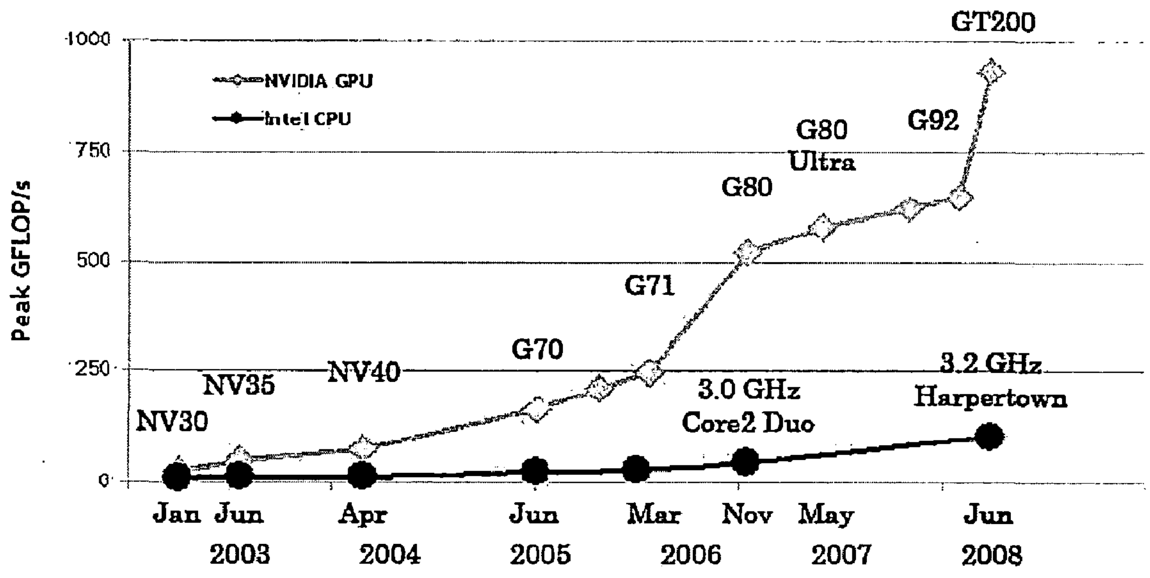

Figure 3.7 Floating point operations for the CPU and the GPU [30]

3.2.1 Programming Model

In this part we will discuss aspects that will explain how the CUDA programming model works and what the various aspects of the model are

3.2.1.1 Thread Hierarchy

Threads in CUDA are arranged in the form of a hierarchy. A number of threads house within what is known as a thread block. These thread block can be 1 dimensional, 2 dimensional or 3 dimensional. These thread blocks are placed in a structure known as thread grid. Thread grid can be either 1 dimensional or 2 dimensional.

A maximum of 512 threads can be placed in a thread block. Thread block are expected to run independently of each other. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling scalable code to be written. Proper selection of grid size and block size is important to gain good speed up.

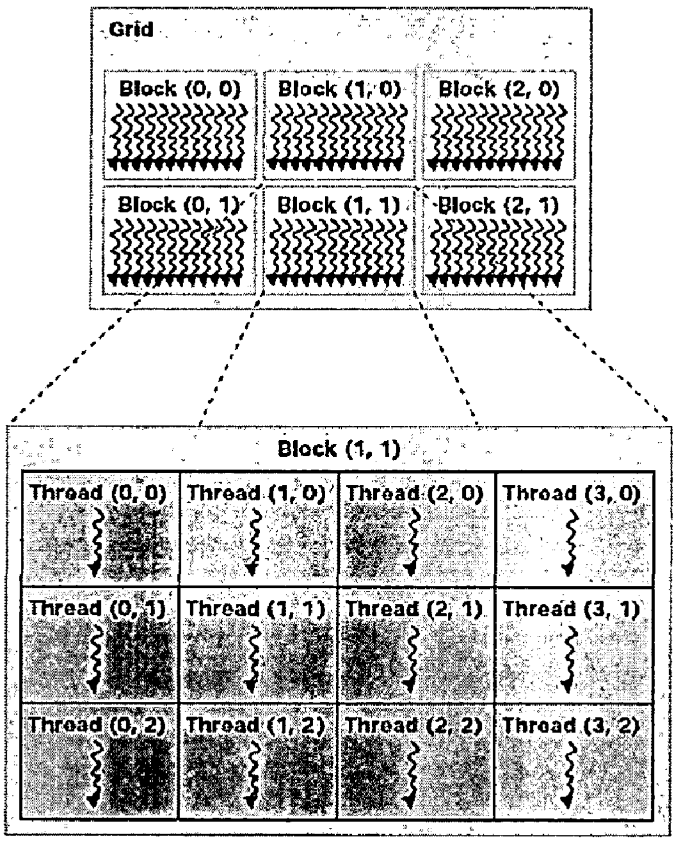

Figure 3.8 Figure showing arrangement of threads [30]

3.2.1.2 Memory Hierarchy

Threads may access memory from different memory spaces during their existence. Threads may declare local variable, may share memory with other threads that belong to the same block or may be accessing global memory.

3.2.2 GPU Implementation

In November 2006 nVidia significantly extended the GPU beyond graphics. It made available the massively parallel multithreaded GPU for general purpose applications. By scaling the number of processors and memory nVidia made available a wide range of products from the high ended GTX 280 with 240 cores and 1 GB RAM to 8400M GS with 16 cores and 128 Megabytes of RAM. The computing features enable a straightforward parallelization of the application by using C language. Some extensions have been made to C for CUDA specific code.

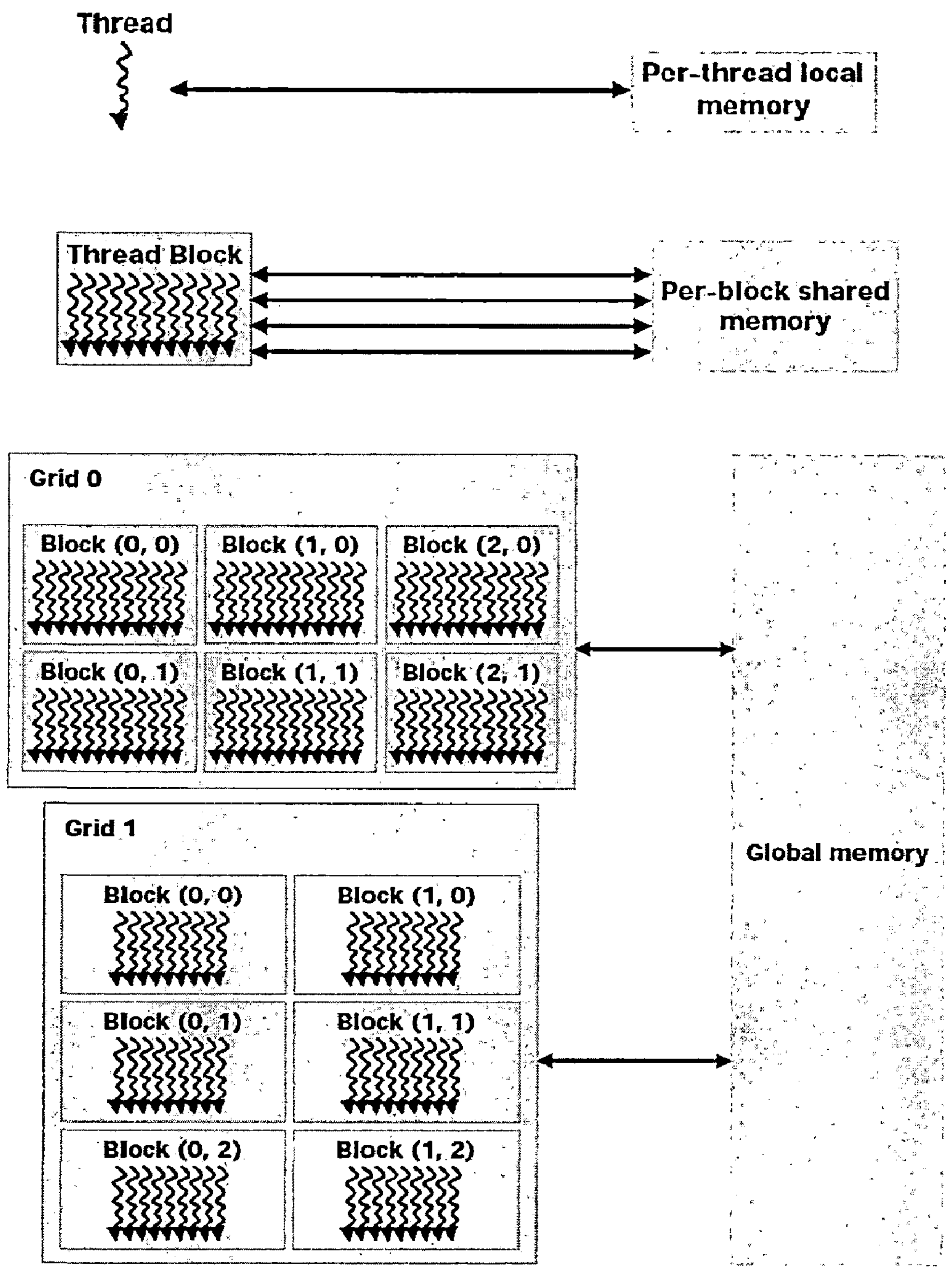

Figure 3.9 Figure showing how threads access global, shared and local memory [30]

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor consists of eight Scalar Processor (SP) cores. Every multiprocessor has 8192 registers of 32 bit size each. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. The general idea is to achieve very fine grained parallelism by assigning one thread to work on one data item. A data element could be a pixel of an image or a protein base when working with poly peptide chains.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address but are otherwise free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it splits them into warps. The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths.

A multiprocessor can work on a maximum of 8 thread blocks. However, if the thread code required a large number of registers then lesser number of thread blocks are assigned to a multiprocessor. In case a thread block is too bulky to be assigned to a multiprocessor then the kernel simply fails to launch.

Chapter 4
Proposed Cover Tree Construction on CUDA GPU

4.1. Challenges in cover tree model-building on CUDA GPU

From the description of cover tree in previous chapter it is clear that cover tree model-building is a computationally intensive task. Many tree data structures have been implemented on CUDA GPU and on other parallel architectures but in cover tree data structure, we cannot implement different parts of tree simultaneously, due to heavy data dependency. In any level i, the points must be separated from each other by distance 2^i and hence the points to be inserted are dependent on each other. Due to this data dependency, data point insertion cannot be done in parallel. Still inserting a point in the tree is a computationally intensive task due to such dependencies on other data points. Before inserting a point we need to make sure that it is well separated from other elements in the same level and this is where the possibilities of parallelization of cover tree algorithm can be explored to exploit the massive computation power of GPU.

The challenges in implementing a parallel algorithm for cover tree were as follows:

- Recursive algorithms cannot be implemented over CUDA platform. Existing algorithm for cover tree construction has recursion and no iterative solution exists before this work. Tail recursions are easy to remove but the recursion in case of cover tree is not a tail recursion therefore to remove the recursion, a different strategy is needed.
- Data dependency is needed to be removed for any algorithm to get good performance on CUDA platform. Due to the separation property of cover tree its data is heavily dependent on each other as explained earlier.

To deal with these challenges we have proposed an iterative strategy to construct the cover tree. With this, in this section we are proposing a variant of cover tree in order to deal with data dependency in cover tree model building process, so that a faster version of cover tree on multicore architectures can be implemented. Also the proposed variant of cover tree is easy to construct and less computation intensive due to some relaxation in separation property.

4.2 Iterative Algorithm for Cover Tree

The primary task before implementing a parallel algorithm for cover tree creation is to remove the recursion.. Following algorithm shows the procedure and the algorithm works as follows:

Algorithm: Iterative algorithm for Cover Tree Construction (CTC)

```
1)  p ← first element of S
2)  Find d_set = {d(p, q); q ∈ S}
3)  root ← p
4)  Stack[0] ← S
5)  d = max {d(insert_ele, q); ∀ q ∈ S}
6)  While (Stack != empty)
7)      S ← Stack[index -- ]
8)      While(S != empty)
9)          insert_ele ← p
10)         While (insert_ele != NULL)
11)             if ( d (insert_ele, p) <= max_dist )
12)                 Remove insert_ele from S
13)                 if( p != insert_ele )
14)                     find d(insert_ele, $q_j$), ∀$q_j$ ∈ S
15)                 new_point_set ← pointer to first element in S
16)                 while( new_point_set != NULL )
17)                     find far_set = { $q_j$; d($q_j$, insert_ele) > 2^current level, ∀ $q_j$ ∈ S}
18)                     d_far = max {d(q, insert_ele); ∀ q ∈ S}
19)                     if (d_far  = 0)
20)                         insert all elements of S as leaf
21)                         insert_ele = NULL
22)                     Else
```

```
23)                 insert insert_ele as child
24)                 parent = insert_ele
25)             Stack[index++] = far_set
26)         else
27)             insert_ele = next element of S
28)     if(Stack.index >1)
29)         merge Stack[index] with Stack[index – 1]
30)         Stack.index - -
```

In this algorithm we are using a stack data structure which keeps track of different data sets at different levels. First the data points are read from the input file and stored as two dimensional array or vectors. Initially the level at which root of the tree is to be inserted is find out using the formula given in equation (1).

$$\text{Level} = \log_2(\text{maxd}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

Where 'maxd' can be defined as:

$$\text{maxd} = \max\{d(\text{root}, q);\ \forall\, q \in S \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

i.e. distance between the root element (here the first element is inserted as root initially) and the farthest element of data set S. "maxd" is calculated at line (5) of algorithm. The complete data set is inserted as first element of stack as shown by line (4). Now the loop at line (6) works on datasets in stack and the number of elements in stack increases when at each level the far set, calculated on the basis of separation distance i.e. 2^i, where i is current level, is inserted onto the stack. Loop at line (10), first tests the element of S for insertion in current level and for that it tests the distance of element from parent in current context with 2^{level} and if the distance is larger than 2^{level} then it looks for another element from S but since we have calculated the level according to maximum distance therefore all the elements can be inserted as child of root in case if all elements are very separated from each other and hence the algorithm will terminate. If distance is less than 2^{level} then we search for near and far elements in S in line (17), and hence far set is kept in the Stack for further processing which will be done later. The loop goes with the elements near (not separated by required distance) to the element to be inserted (insert_ele in algorithm). Eventually we remain with either same elements as insert_ele or only with insert_ele and in this case line (19 – 21) inserts all elements as children. Now the last far set inserted in Stack is merged with the far set inserted previous to this set in line (21) because these elements have already been tested and hence they had been kept in far set. At this point we take last merged set in S and continue with the same procedure. The algorithm terminates when all elements kept in the Queue are processed. The distance measured at all points is Euclidean distance. Figure 4.1 shows a simple flow diagram of this algorithm.

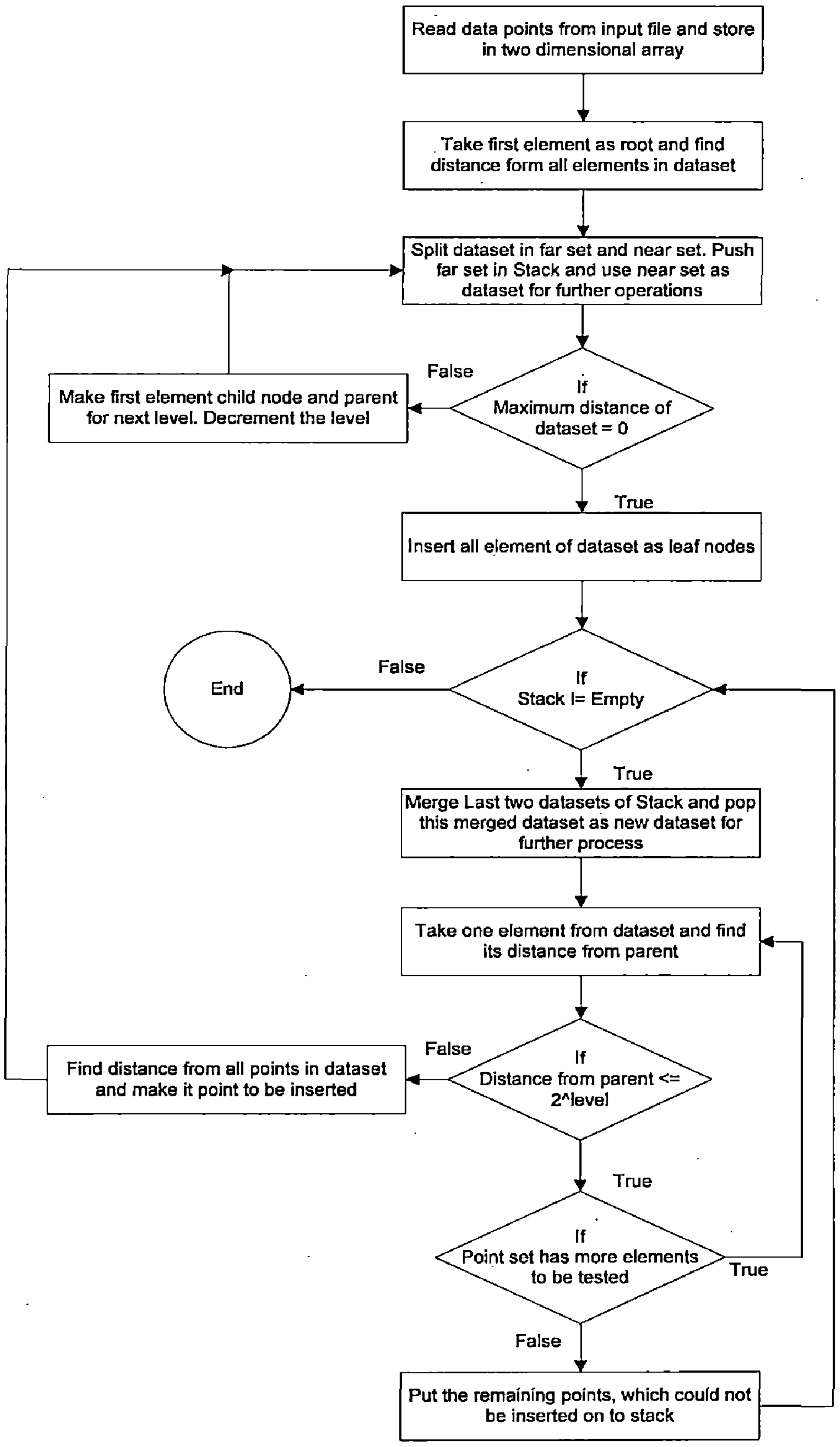
Read data points from input file and store in two dimensional array
Take first element as root and find distance form all elements in dataset
Split dataset in far set and near set. Push far set in Stack and use near set as dataset for further operations
If Maximum distance of dataset = 0
False
Make first element child node and parent for next level. Decrement the level
True
Insert all element of dataset as leaf nodes
If Stack != Empty
False
End
True
Merge Last two datasets of Stack and pop this merged dataset as new dataset for further process
Take one element from dataset and find its distance from parent
If Distance from parent <= 2^level
False
Find distance from all points in dataset and make it point to be inserted
True
If Point set has more elements to be tested
True
False
Put the remaining points, which could not be inserted on to stack

Figure 4.1 Flow diagram of iterative algorithm for cover tree creation

4.3 GPU Cover Tree Construction (GCTC) Algorithm

The parallel algorithm of cover tree is given in following algorithm and it is very similar to the iterative algorithm explained before. All the computationally intensive tasks are done simultaneously by many threads at line (7 – 9), (24 – 25), (30 – 32). These are the places where we get advantage of multicore architecture and implementation gains the speedup over existing linear implementation for single core CPU.

Algorithm: Graphics Cover Tree Construction (GCTC) Algorithm

```
1)   parent ← first element of S
2)   root ← parent
3)   Queue[0] ← S
4)   Total_threads ← blockDim.x * threadDim.x
5)   Call the GPU kernel
6)     tid = blockIdx.x * blockDim.x + threadIdx.x
7)     for(i = tid; i < S.index; i += Total_threads)
8)       d = max {d(root, q_i); q_i ∈ S}
9)     d = max {d(root, q); q ∈ S}
10)    While (Queue != empty)
11)      synchronize all threads
12)      if(tid = 0)
13)        S ← Queue[index -- ]
14)      while(S != empty)
15)        if(tid = 0)
16)          insert_ele ← p
17)        Synchronize all threads
18)        While (insert_ele != NULL)
19)          if(tid = 0)
20)            if (d (insert_ele , parent) <= max_dist)
21)              Remove insert_ele from S
22)          Synchronize all threads
23)          if(parent != insert_ele)
```

```
24)                     for( j = tid; j < S.index; j += Total_threads)
25)                         find d(insert_ele, $q_j$);  $\forall q_j \in S$
26)                     if(tid = 0)
27)                         new_point_set ← pointer to first element in S
28)                     synchronize all threads
29)                     while(new_point_set != NULL)
30)                         for(j = tid; j < S.index; j += Total_thread)
31)                             find far_set = { $q_j$ ; d ($q_j$, insert_ele) > 2^current level, $\forall q_j \in S$}
32)                         d_near = max {d(q, insert_ele); $\forall q \in S$}
33)                         if (d_near  = 0)
34)                             insert all elements of S as leaf
35)                             S ← Empty
36)                             insert_ele = NULL
37)                         Else
38)                             if(tid = 0)
39)                                 insert insert_ele as child
40)                                 parent = insert_ele
41)                 if(tid = 0)
42)                     Queue[index++] = far_set
43)             Else
44)                 if(tid = 0)
45)                     insert_ele = next element of S
46)         if(Queue.index >1)
47)             if(tid = 0)
48)                 merge Queue[index] in Queue[index – 1]
49)                 Queue.index - -
50)         synchronize all threads
51) Copy tree from device to host
```

4.4 Cover Tree Variant Construction (CTVC) Algorithm

The new proposed algorithm is a variant of cover tree algorithm and it introduces a change in third property of cover tree i.e. separation property. While designing the cover tree batch insertion algorithm for CUDA platform, the main problem faced is that, we cannot implement different parts of tree simultaneously because of data dependency due to separation property which says that at each level the distance between each point must be greater than 2^i where 'i' is that level. If we start batch insertion of cover tree in many parts then we cannot maintain the separation property. While designing the algorithm it was observed that if we apply separation property locally for children of each node than also this tree maintains all other characteristics.

The same nearest neighbour search algorithm can still be used and we get an advantage that now different parts of tree can be constructed simultaneously. The iterative algorithm for this variant is given in following algorithm and an example is shown in figure 4.2 for dataset {1, 3, 4, 0, 2, 8, 9, 4, -1, 1, 2, and 8}. Figure 4.3 shows the flow diagram of the variant of cover tree. The iterative variant algorithm for single core architecture works as follows:

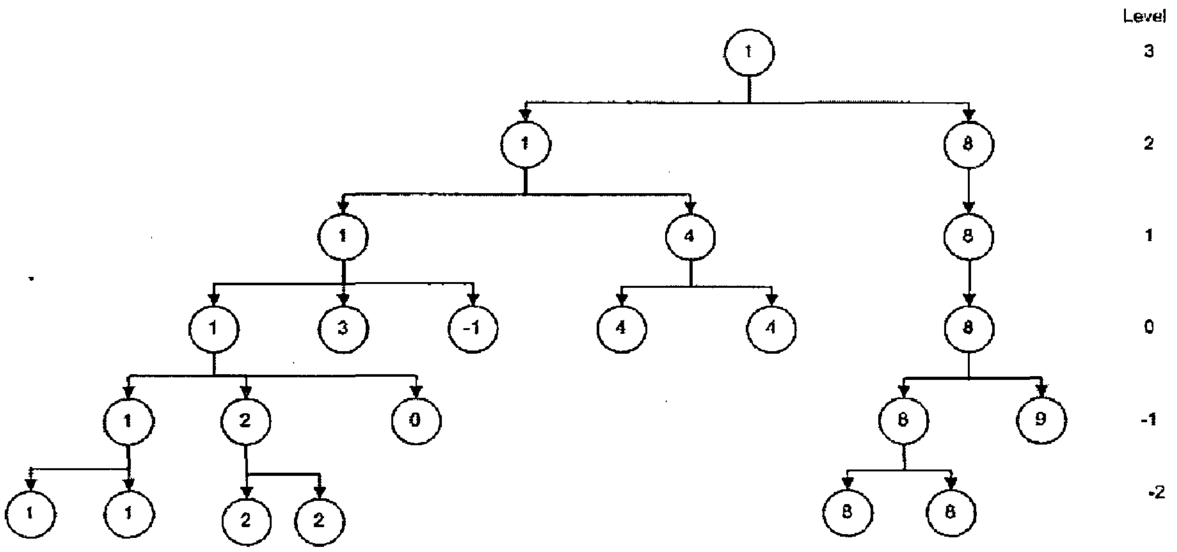

Figure 4.2 Tree structure of proposed variant of cover tree

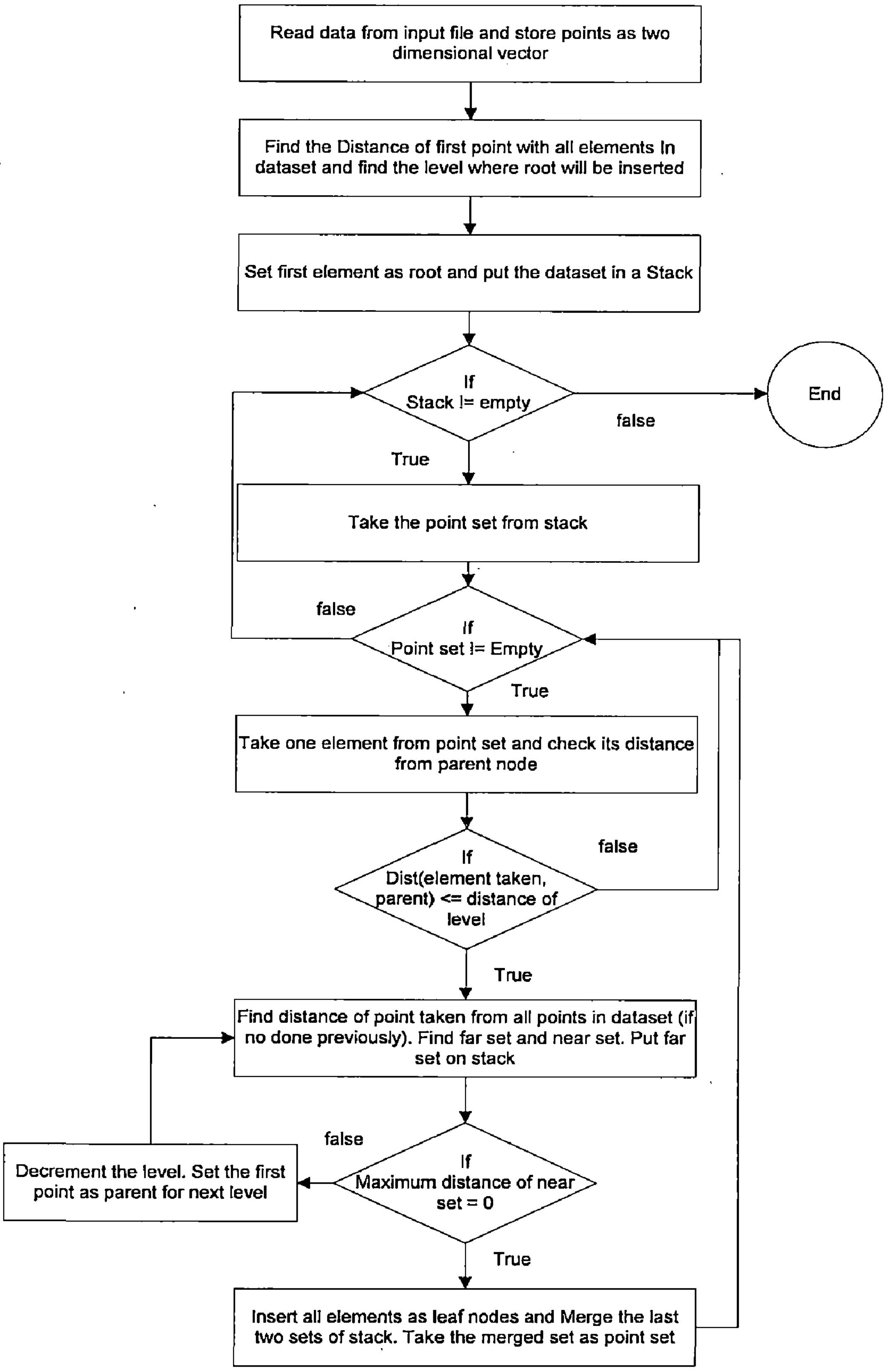
Read data from input file and store points as two dimensional vector
Find the Distance of first point with all elements In dataset and find the level where root will be inserted
Set first element as root and put the dataset in a Stack
If Stack != empty
false
End
True
Take the point set from stack
false
If Point set != Empty
True
Take one element from point set and check its distance from parent node
false
If Dist(element taken, parent) <= distance of level
True
Find distance of point taken from all points in dataset (if no done previously). Find far set and near set. Put far set on stack
false
Decrement the level. Set the first point as parent for next level
If Maximum distance of near set = 0
True
Insert all elements as leaf nodes and Merge the last two sets of stack. Take the merged set as point set

Figure 4.3 Flow diagram of cover tree variant construction algorithm

The first element is inserted as root of tree and distance of this element is found with all the elements in the dataset S as done in line (3-4). Now in line (11 - 20) the algorithm iteratively finds the far set and near set while going deep down in the tree simultaneously inserting the element to be inserted until in dataset S no element remains or the element with distance zero with the element being inserted remains and at this point all these elements are inserted as leaf in tree as shown in line (12 -14). The far sets are kept in Queue and then each far set is processed in line (7) of algorithm. We have tested the new algorithm for nearest neighbour and same results are obtained for nearest neighbour search.

Algorithm: Iterative algorithm for Cover Tree Variant Construction (CTVC)

```
1)  Queue[index++] = S
2)  root ← first element of S
3)  insert_ele ← root
4)  find d = {d(root, q); ∀q ∈ S}
5)  d_max = max(d)
6)  top_level = log2 d_max
7)  while(Queue != empty)
8)      point_set ← Queue[index ]
9)      parent ← Queue[index ]
10)     insert_ele ← point_set[0]
11)     while(point_set != empty)
12)         if(d_max = 0)
13)             insert q as leaf, ∀ q ∈ point_set
14)             point_set ← NULL
15)         else
16)             find far_set = {q ; d(insert_ele, q) > 2^ top_level, ∀ q ∈ point_set}
17)             point_set = point_set – (point_set ∩ far_set)
18)             Queue[index ++] = far_set
19)             insert insert_ele as a child
20)             Parent ← insert_ele
21) return root
```

4.5 GPU Cover Tree Variant Construction (GCTVC) Algorithm

The parallel version of the CTVC algorithm works in quite similar manner. Here, as the far sets are created and kept in Queue then they are processed by another block of threads. We have assigned a block of thread to process each data set so that we can get advantage of shared memory area within each block.

Algorithm: Graphics Cover Tree Variant Construction (GCTVC) Algorithm

```
1)  Queue[index++] = S
2)  root ← first element of S
3)  insert_ele ← root
4)  total_thread = blockDim.x * threadDim.x
5)  Call the kernel
6)      tid = blockIdx.x * blockDim.x + threadIdx.x
7)      for(i = tid; i < S.index; i += total_thread)
8)          find d = {d(root, q_i); ∀ q_i ∈ S}
9)      d_max = max(d)
10)     if(threadIdx.x = 0)
11)         top_level = log2 d_max
12)     while(Queue != empty)
13)         for(j = blockIdx.x; j < Queue.index; j += blockDim.x)
14)             if(threadIdx.x = 0)
15)                 point_set ← Queue[j ]
16)                 parent ← parent.Queue[j ]
17)                 insert_ele ← point_set[0]
18)             synchronize threads of current block
19)             while(point_set != empty)
20)                 if(d_max = 0)
21)                     for(i = threadIdx.x; i < point_set.index; i += blockDim.x)
22)                         insert q_i as leaf, ∀ q_i ∈ point_set
23)                     if(threadIdx.x = 0)
24)                         point_set ← NULL
```

```
25)             else
26)                 find far_set = {q ; d(insert_ele, q) > 2^ top_level, ∀ q ∈ point_set}
27)                 if(threadIdx.x = 0)
28)                     point_set = point_set – (point_set ∩ far_set)
29)                     Queue[index ++] = far_set
30)                     insert insert_ele as a child
31)                     Parent ← insert_ele
32)             synchronize all threads of current block
33) copy tree.index elements of tree from device to host
```

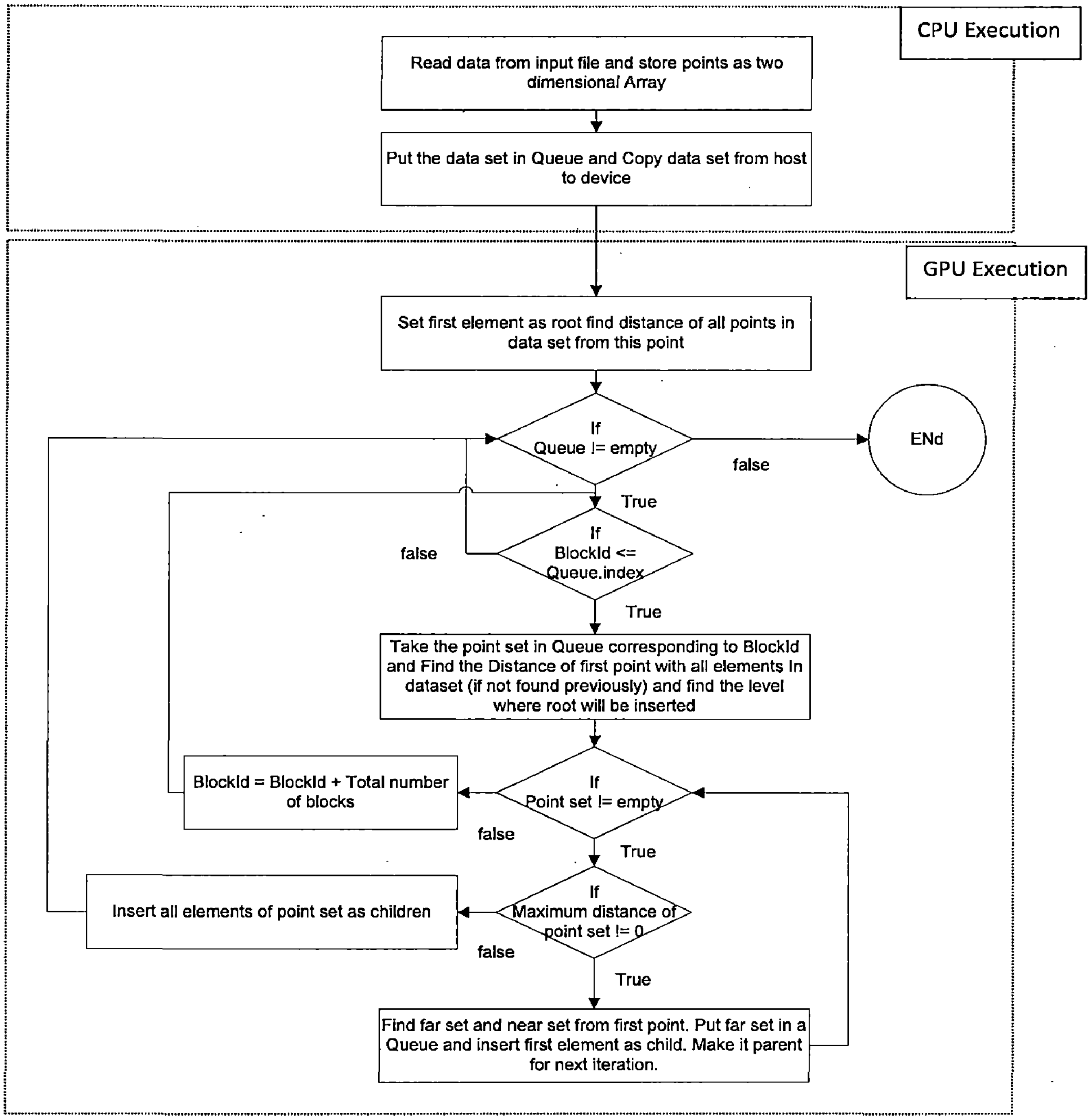
CPU Execution
Read data from input file and store points as two dimensional Array
Put the data set in Queue and Copy data set from host to device
GPU Execution
Set first element as root find distance of all points in data set from this point
If Queue != empty
false
ENd
True
If BlockId <= Queue.index
false
True
Take the point set in Queue corresponding to BlockId and Find the Distance of first point with all elements In dataset (if not found previously) and find the level where root will be inserted
BlockId = BlockId + Total number of blocks
If Point set != empty
false
True
Insert all elements of point set as children
If Maximum distance of point set != 0
false
True
Find far set and near set from first point. Put far set in a Queue and insert first element as child. Make it parent for next iteration.

Figure 4.4 Flow diagram of GCTVC Algorithm

Chapter 5
Results and Discussions

We measured the relative performance of our algorithm of cover tree by comparing the execution time of our GPU version of cover tree code and its single core version. The test machine has 8 intel Xeon processors, each working on 2.00 GHz frequency, having 6144 KB of cache and 3 GB of RAM. The machine has an NVIDIA fx 4600 graphics card which has 112 cuda cores, and total 768 MB of GPU memory with bandwidth 67.2 GB/sec. The machine was running fedora core 11. The dataset for performance measurement purpose was taken from UCI ML repository [31]. The data files are from different fields such as covtype is dataset for prediction of forest cover type from cartographic variables, bio_train and bio_test are the datasets from molecular biology related to gene sequences, phy_test and phy_train are the datasets of physical symptoms of heart disease patients. Table 5.1 shows the runtime statistics and data description for existing cover tree algorithm for single core CPU. Table 5.2 and table 5.3 shows the results and performance improvement of GCTC and GCTVC algorithm respectively over single core CPU execution of cover tree algorithm for different number of thread blocks and each block had 256 threads.

5.1. Runtime Statistics for Existing Linear Algorithm

Table 1: Runtime statistics of existing single core CPU algorithm

File Name	File Size (MB)	Dimension of points	Number of data points in file	Tree construction time for serial code (sec)
Phy_train	48.8	79	50000	1.1313
Phy_test	97.3	78	100000	2.3807
covtype	71.1	55	581012	6.3826
Bio_train	65	76	135908	7.9179
Bio_test	62	74	139658	8.4430

5.2. Runtime Statistics for GCTC Algorithm

Table 3: Runtime statistics of GCTC algorithm

File Name	Time in Sec. for different number of blocks for GCTC algorithms				
	1 Block	2 Blocks	16 Blocks	64 Blocks	Maximum Improvement (%)
Phy_train	1.4706	1.3515	0.4262	0.4261	62.33
Phy_test	3.0949	2.8568	0.8116	0.8114	65.91
Covtype	8.2973	7.6591	1.6720	1.6720	73.80
Bio_train	10.2932	9.5014	2.4428	2.4420	69.15
Bio_test	10.975	10.0320	2.9598	2.8667	67.89

5.3. Runtime Statistics for GCTVC Algorithm

Table 2: Runtime statistics of GCTVC algorithm

File Name	Time in Sec. for different number of blocks for GCTVC algorithms				
	1 Block	2 Blocks	16 Blocks	64 Blocks	Maximum Improvement (%)
Phy_train	1.4706	1.3517	0.3399	0.3031	73.20
Phy_test	3.0949	2.8569	0.7152	0.5553	76.67
covtype	8.2973	7.6597	1.5102	1.2770	79.99
Bio_train	10.2932	9.5018	2.2131	1.7720	77.62
Bio_test	10.975	10.1320	2.2798	1.9907	76.42

5.4. Performance Comparison of Linear, GCTC and GCTVC Algorithms

The comparisons of time required for Cover Tree construction between linear algorithm and the two algorithms GCTC and GCTVC are shown in figure 5.1 and figure 5.2 respectively. The performance comparison of GCTC and GCTVC algorithms for different numbers of blocks of threads is shown by comparing the time to construct the tree for different files using the two algorithms. This comparison is shown using the graphs shown by figure 5.3 to figure 5.7 for different files. The figures shows that with small number of threads the performance of two algorithms is quite similar but as the number of blocks of threads increases GCTVC algorithm provides better parallelization as compare to GCTC algorithm.

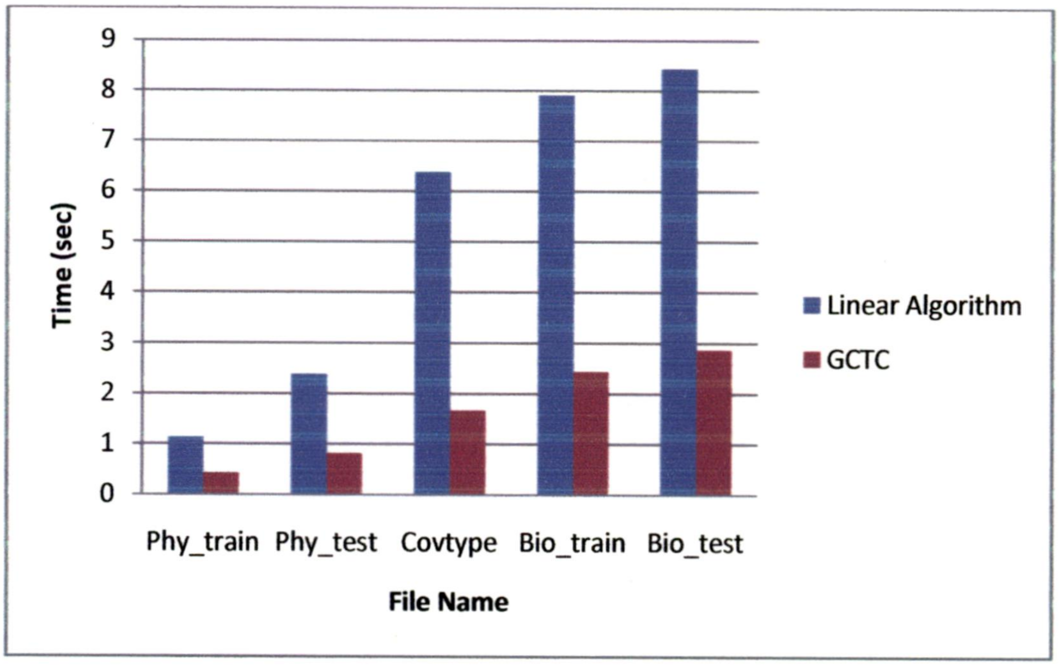

Figure 5.1 Comparison of Time between Linear and GCTC Algorithms

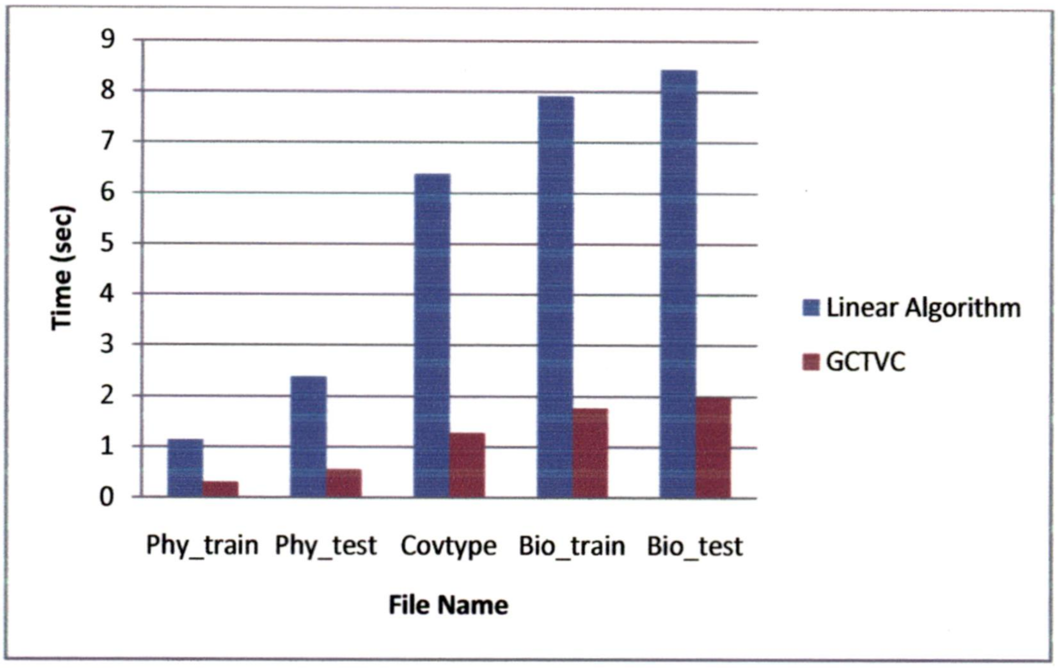

Figure 5.2 Comparison of Time between Linear and GCTVC Algorithms

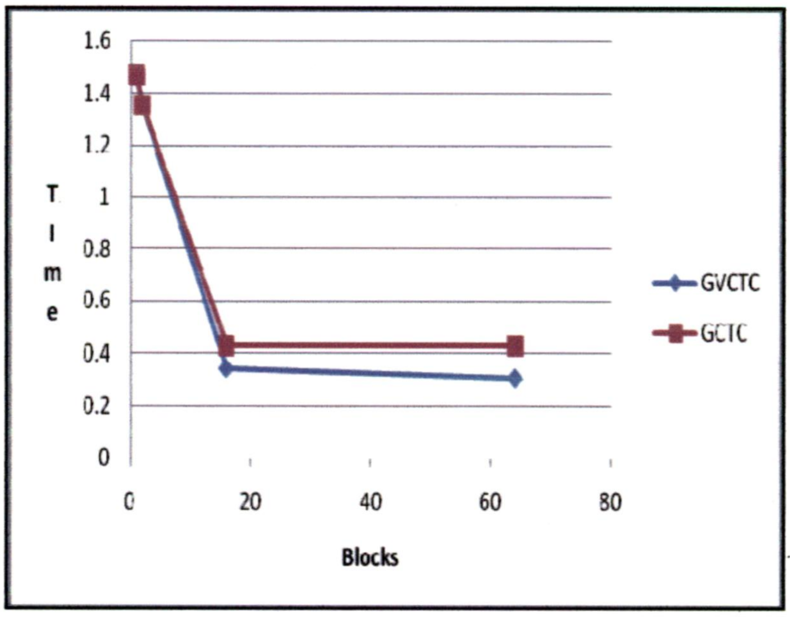

Figure 5.3 Comparison of GCTC and GCTVC algorithms for file Phy_train

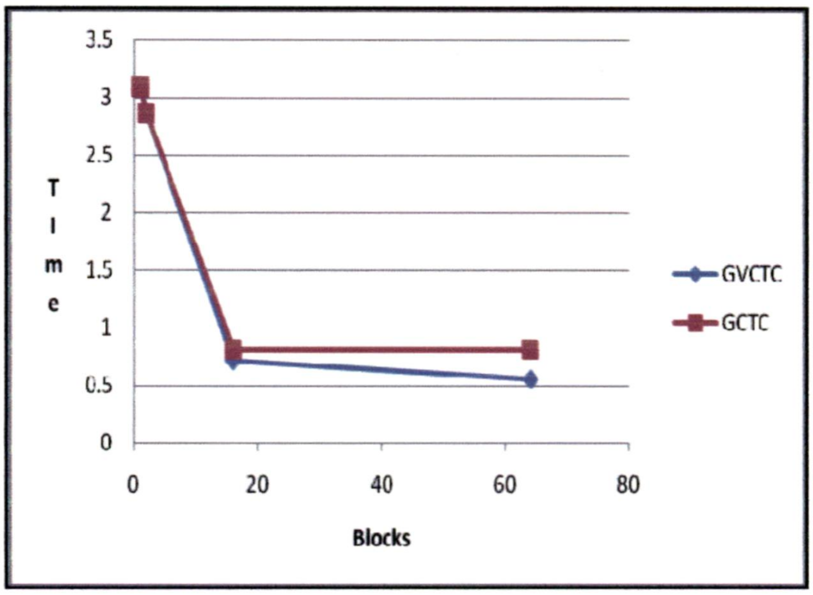

Figure 5.4 Comparison of GCTC and GCTVC algorithms for file Phy_test

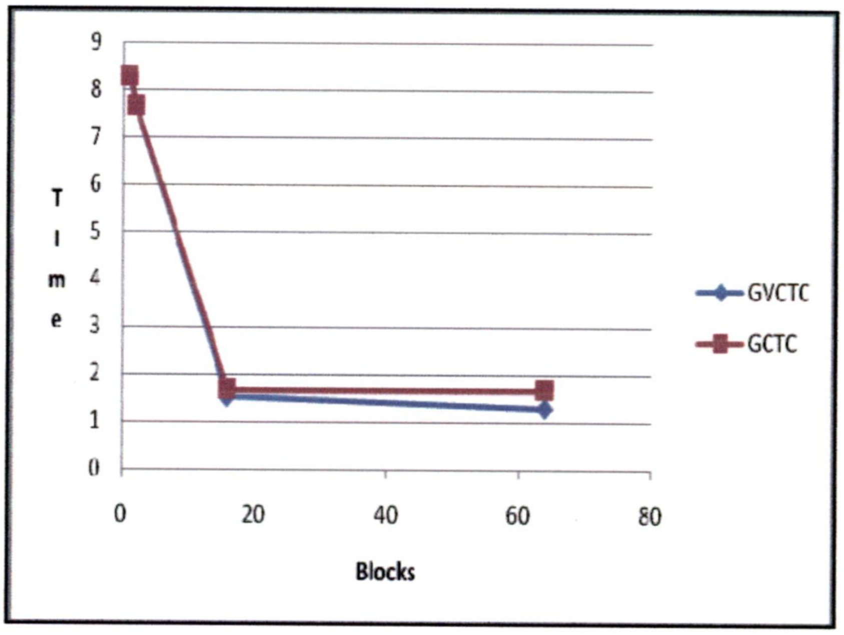

Figure 5.5 Comparison of GCTC and GCTVC algorithms for file covtype

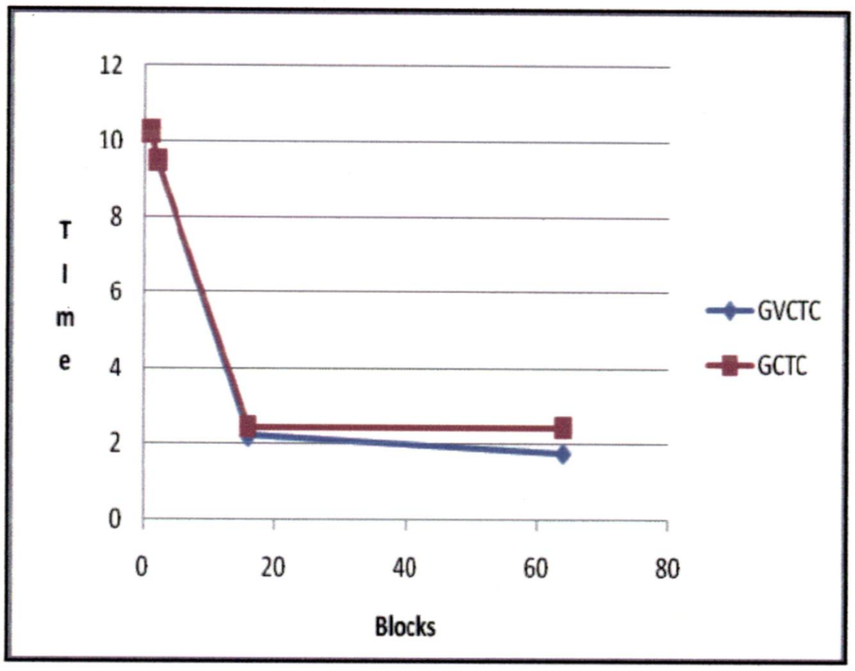

Figure 5.6 Comparison of GCTC and GCTVC algorithms for file Bio_train

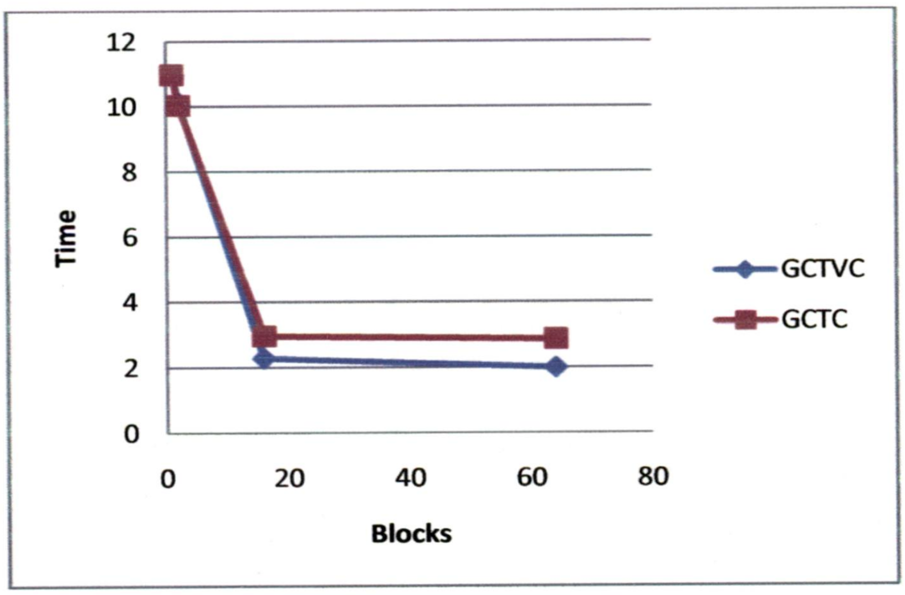

Figure 5.7 Comparison of GCTC and GCTVC algorithms for file Bio_test

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we showed how multicore processors can be used for solving computationally intensive algorithmic problems. We gave an approach to speed up the process of model-building for nearest neighbour search using CUDA GPU.

We implemented the proposed parallel algorithm for cover tree construction on CUDA platform. This approach facilitates implementation of algorithms with data dependencies, yet computationally intensive. We also proposed a variant of cover tree which is easier to construct and less computationally intensive as compare to cover tree also it gives a better parallelization on CUDA GPU.

Following conclusions can be made from the results obtained using the proposed algorithm of cover tree implementation over CUDA GPU and the variant of cover tree over CUDA GPU on the aforementioned data:

- The proposed GPU cover tree construction algorithm and the variant of cover tree algorithm on CUDA GPU can give better performance by speeding up the cover tree model building process by approximately *three* and *five* times respectively.
- The proposed model can be used for other distance based model-building processes which are computationally intensive.
- The proposed algorithms outperform the existing linear implementation of cover tree model building algorithm on single core CPU architecture.
- The proposed variant of cover tree is simple to implement and less computation intensive in terms of model building process time.
- The proposed models are highly suitable for applications with dynamic dataset which needs to find neighbours.
- Advantages of multicore systems lie not only in performance improvement but also in terms of cost effectiveness and resource utilization. Thus, the trend of using multicore systems for solving computationally intensive problems can be

viewed as a simple and highly beneficial means for performance improvement. With the use of cover tree construction on multicore processors various applications that require such model building in real time can be successfully deployed.

6.2 Future Work

There is significant room for improvement in our work for fast model-building process of cover tree and other similar model-building processes. The possible improvements are listed below:

- There are other parallel processing multicore architectures such as CELL BE, which can work with less communication cost and hence further performance improvement may be achieved.
- There are some places in the parallel algorithms presented here, where synchronization between threads has been used and this reduces the performance of parallel algorithm. These synchronizations can be removed for better performance than the solution presented here.
- The nearest neighbour search algorithm using cover tree can be implemented on multicore architecture so that the overall nearest neighbour search cost can be reduced and the model will become more suitable for dynamic dataset.
- Other classification and clustering problems which are computation intensive and require model-building, can be improved using the same approach.
- The algorithms given in this thesis can be optimized further for better results.

Since we are rapidly moving towards the automation of everything, there will be huge requirement for fast nearest neighbour search using dynamic datasets.

REFERENCES

[1] P. Legendre, "Reconstructing Biogeographic History Using Phylogenetic- Tree Analysis of Community Structure," *Systematic Zoology*, vol. 35, pp. 68—80, 1986.

[2] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 6, pp. 563-580, 1980.

[3] K. Clarkson, "Nearest neighbour queries in metric spaces," in *Proc. Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 63–93, 1999.

[4] D. Karger and M. Ruhl, "Finding Nearest Neighbours in Growth Restricted Metrics," in *Proc. 34th annual ACM symposium on Theory of computing*, pp. 74-750, 2002.

[5] R. Krauthgamer and J. Lee, "Navigating Nets: Simple Algorithms for Proximity Search," in *Proc. 15th Annual Symposium on Discrete Algorithms (SODA)*, pp. 791-801, 2004.

[6] A. Beygelzimer, S. Kakade, and J. Langford, "Cover Trees for Nearest Neighbour," in *Proc. 23rd international conference on Machine learning*, pp. 97—104, 2006.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp.80-113, 2007.

[8] P. Indyk, and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality" in *Proc. thirtieth Annual ACM Symposium on Theory of Computing*, pp. 604-613, 1998.

[9] B. Babcock, M. Datar, and R. Motwani, "Load Shedding for Aggregation Queries over Data Streams," in *Proc. 20th International Conference on Data Engineering, (ICDE)*, pp. 350-361, 2004.

[10] A. Gersho, and R. M. Gray, "VECTOR QUANTIZATION AND SIGNAL COMPRESSION," Kluwer Academic Publishers (USA), 1991.

[11] T. M. Cover, and P. E. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21-27, 1967.

[12] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally Weighted Learning," *Artificial Intelligence Review,* vol. 11, no. 1-5, pp. 11-73, 1997.

[13] S. Niijima, S. Kuhara, "Effective Nearest Neighbor Methods for Multiclass Cancer Classification Using Microarray Data," in *Proc. 16th International Conference on Genome Informatics*, pp. 57-59, 2005.

[14] J. F. Robinson-Cox, M. M. Bateson, and D. M. Ward, "Evaluation of Nearest Neighbor Methods for Detection of Chimeric Small-subunit rrna Sequences," *Applied and Environmental Microbiology*, vol. 61, no. 4, pp. 1240-1245, 1995.

[15] S. Dudoit, J. Fridlyand, and T. P. Speed, "Comparison of Discrimination Methods for the Classification of Tumors Using Gene Expression Data," *Journal of the American Statistical Association,* vol. 97, no. 457, pp. 77-87, 2002.

[16] T. Li, C. Zhang, M. Ogihara, "A Comparative Study of Feature Selection and Multiclass Classification Methods for Tissue Classification Based on Gene Expression," *Bioinformatics*, vol. 20, no. 15, pp. 2429-2437, 2004.

[17] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, "Query by Image and Video Content: The Qbic System" *Computer*, vol. 28, no. 9, pp. 23-32, 1995.

[18] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith, "Query by Humming: Musical Information Retrieval in an Audio Database," in *Proc. third ACM international conference on Multimedia,* pp. 231-236, 1995.

[19] Y. Cai, N. Freitas, and J. J. Little, "Robust Visual Tracking for Multiple Targets," in *Proc. 9th European Conference on Computer Vision*, pp. 107-118, 2006.

[20] D. Lucarella, "A Document Retrieval System Based on Nearest Neighbour Searching," *Journal of Information Science*, vol. 14, no. 1, pp. 25-33, 1988.

[21] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391-407, 1990.

[22] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[23] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematics Software*, vol. 3, no. 3, pp. 209-226, 1977.

[24] J. S. Beis, D. G. Lowe, "Shape Indexing Using Approximate Nearest-Neighbour Search," in *Proc. Conference on Computer Vision and Pattern Recognition*, pp. 1000-1006, 1997.

[25] T. Kollar, "Fast Nearest Neighbours," Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Technical report, 2006.

[26] J. Hays and A. A. Efros, "Scene Completion Using Millions of Photographs," *ACM Transactions on Graphics (SIGGRAPH)*, vol. 26, no. 3, 2007.

[27] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proc. fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[28] Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design," *IEEE Transactions on Communications*, vol. 28, no. 1, pp. 84–94, January 1980.

[29] A. W. Fu, P. M. Chan, Y. Cheung, Y. S. Moon, "Dynamic Vp-tree Indexing for n-nearest Neighbor Search Given Pair-wise Distances."*International Journal on Very Large Data Bases*, vol. 9, no. 2, pp. 154–173, 2000.

[30] NVIDIA Corporation: "NVIDIA CUDA COMPUTE UNIFIED DEVICE ARCHITECTURE PROGRAMMING GUIDE," Published: NVIDIA Corporation, Jan 2007.

[31] "UCI Machine Learning Repository," *http://archive.ics.uci.edu/ml*, Last Accessed on 13th June 2010.

LIST OF PUBLICATIONS

[1] Mukesh Sharma, R C Joshi, "Design and Implementation of Cover Tree Algorithm on CUDA-Compatible GPU", *International Journal of Computer Applications*, vol. 3, no. 8, pp. 24 -27, 2010.
DOI: 10.5120/748-1057
URI: http://www.ijcaonline.org/archives/volume3/number8/748-1057

[2] Mukesh Sharma, R C Joshi, "Model Building For Nearest Neighbor Search on CUDA Compatible GPU", in *Proc. International Conference on Advances in Information and Communication Technologies (ICT),* Kochi, Kerala, India, Sep 07-09, 2010. (Accepted for publication in Springer)