International Journal of Computer Applications (0975 – 8887)
Volume 3 – No.8, June 2010

Design and Implementation of Cover Tree Algorithm on CUDA-Compatible GPU

Mukesh Sharma
Department of Electronics & Computer Engineering
IIT Roorkee, Uttarakhand, India

R. C. Joshi
Department of Electronics & Computer Engineering
IIT Roorkee, Uttarakhand, India

ABSTRACT
Recently developed architecture such as Compute Unified Device Architecture (CUDA) allows us to exploit the computational power of Graphics Processing Units (GPU). In this paper we propose an algorithm for implementation of Cover tree, accelerated on the graphics processing unit (GPU). The existing algorithm for Cover Tree implementation is for single core CPU and is not suitable for applications with large data set such as phylogenetic analysis in bioinformatics, in order to find nearest neighbours in real time. As far as we know this is first attempt made ever to implement the cover tree on GPU. The proposed algorithm has been implemented using compute unified device architecture (CUDA), which is available on the NVIDIA GPU. The proposed algorithm efficiently uses on chip shared memory in order to reduce the data amount being transferred between offchip memory and processing elements in the GPU. Furthermore our algorithm presents a model to implement other distance trees on the GPU. We show some experimental results comparing the proposed algorithm with it's execution on pre-existing single core architecture. The results show that the proposed algorithm has a significant speedup as compare to the single core execution of this code.

General Terms
Algorithms

Keywords
Cover Tree, CUDA, GPU, Parallel Algorithm, Multicore architectures.

1. INTRODUCTION
Finding the nearest neighbor of a point in a given metric is a classical algorithmic problem which has many practical applications such as database queries, in particular for complex data such as multimedia, phylogenetic tree analysis [1] for the study of evolutionary relatedness among various groups of organisms in biology. There are many data structures and methods to find the nearest neighbor such as naive implementation, methods given in [2], [3], and [4], Navigating Nets [5], Cover Tree [6]. The Cover Tree data structure is relatively new and it was introduced first by A. Beygelzimer et al. [6] and was proved to be efficient in terms of space complexity as well as time complexity for constructing the tree (preprocessing) and nearest neighbor search as compared to other data structures and methods.

Most personal computer workstations today contain hardware for 3D graphics acceleration called Graphics Processing Units (GPUs). Recently, GPUs feature hardware optimized for simultaneously performing many independent floating-point arithmetic operations for displaying 3D models and other graphics tasks. Thus, GPGPU programming has been successful primarily in the scientific computing disciplines which involve a high level of numeric computation. However, other applications could be successful, provided those 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 general-purpose (GPGPU) applications [7]. These recently developed architectures such as NVIDIA Compute Unified Device Architecture (CUDA) has not been explored for several bio-informatics problems. This architecture is very cheap and requires parallel algorithms.

The cover tree construction is a pre-requirement to find the nearest neighbours. Since in many applications the dataset is very large and the tree construction consumes time therefore in all such cases the speedup in tree construction is highly required to find the closest points in real time. As far as we know this is vary first attempt to make the cover tree insertion algorithm parallel in order to use the processing power of GPUs but many other algorithms have been parallelized in different fields on CUDA and IBM Cell architectures. K. Zhou et. al [8] implemented real time KD-Tree construction algorithm on GPUs and achieved significant speedup. Similar attempt has been made by M. Schatz et. al [9] to implement sequence alignment algorithm to achieve speedup over the well-optimized single core CPU algorithm.

Serialization has a limit and it cannot be improved further but parallelization is a way to make computationally intensive algorithm more efficient. Cover Tree is one of the data structures which has not been parallelized to exploit the computation power of GPUs. With this, the existing algorithm for cover tree implementation was having recursion and no previous attempts were made to remove the recursion. We have removed the recursion from the existing algorithm for single core CPU and also proposed an algorithm for cover tree to construct it on using graphics hardware. The algorithm has been tested on NVIDIA graphics card.

2. BACKGROUND
Before proceeding further, we give a brief introduction about cover tree. A cover tree T on a dataset S is a leveled tree. Each level is index by an integer scale i which decreases as the tree is descended. A node in the tree may have many children node but it can have only one parent node. 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)

(2) $\forall p \in C_{i-1}$, there exists a q C_i such that $d(p, q) \leq 2i$ and there is exactly one q that is a parent of p. (covering tree)

(3) ∀p, q ∈ Ci, d(p, q) > 2i (separation)

In a cover tree each level is a cover for the level above it i.e. if a point is present in level Ci, then it will also be in all the levels below it. This is the first property nesting (see Figure 1).

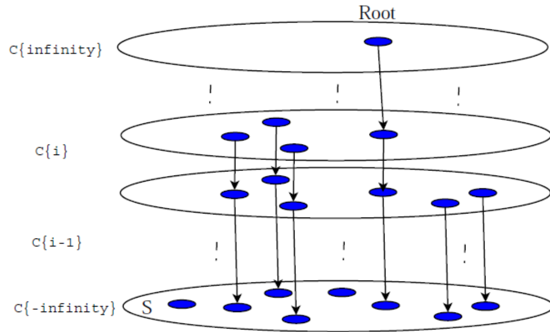

Figure 1 Graphical representation of Nesting Property [10]

Each node has only one parent and it's distance from parent node is less than 2i. This is called covering property (see Figure 2).

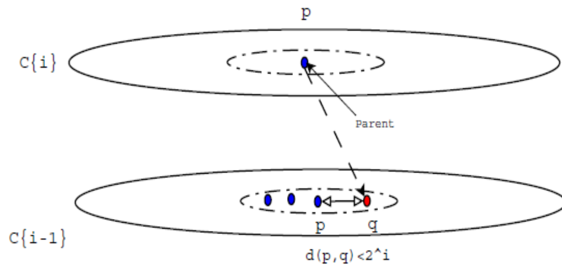

Figure 2 Graphical representation of covering tree Property [10]

At each level i, the points at that level are 2i distance apart from each other (see Figure 3).

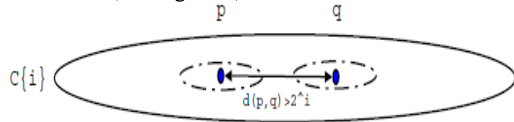

Figure 3 Graphical representation of separation Property [10]

3. RELATED WORK

Many parallel algorithms have been designed to gain speed up over their single core implementations.

3.1 Batched GPU algorithm for set intersection

Di Wu et. al [11] implemented the parallel algorithm to find intersection of inverted lists . The algorithm feeds queries to GPU in batches, thus it takes full advantage of GPU processor cores even if problem size is small. The algorithm also proposes an input preprocessing method which alleviate load imbalance effectively. Their experimental results show that the batched algorithm is much faster than the fastest CPU algorithm and plain GPU algorithm. In order to make hundreds of GPU shaders busy the algorithm pumps the query to GPU in batches instead of sending them one by one. CPU is in charge of only task scheduling and data transferring. All calculative tasks are offloaded to GPU.

3.2 Sequences alignment using GPU

M. Schatz et. al [9] proposed an algorithm MUMmerGPU to implement high-throughput sequence alignment algorithm mummer on GPU. The algorithm performs parallelized exact string alignment on the GPU. First a suffix tree is created on CPU and of the reference sequence is constructed on the CPU and transferred to the GPU. Then the query sequences are transferred to the GPU, and are aligned to the tree on the GPU using an alignment algorithm. Alignment results are temporarily written to the GPU's memory, and then transferred in bulk to host RAM once the alignment kernel is complete for all queries. Finally, all maximal alignments longer than a user-supplied value are reported by post- processing the raw alignment results on the CPU. Thus by transferring most of the computational work on GPU the algorithm achieved significant speedup in performance. The results show that a significant speedup, as much as a 10-fold, can be achieved through the use of cached texture memory and data reordering to improve access locality.

The above related works show that many algorithms have been implemented for execution on GPUs by offloading the computational part of the algorithm on GPUs and significant speedup has been achieved.

4. GPU COVER TREE CONSTRUCTION (GCTC)

The primary task before implementing a parallel algorithm for cover tree creation is to remove the recursion. We have adopted a similar approach to BFS to construct the tree iteratively. The following algorithm 1 shows the procedure:

Algorithm 1: Iterative algorithm for Cover Tree Construction (CTC)

1) Set Q_1 = { q : q ∈ Dataset }
2) Root = first element of Q_1
3) Set root as parent for the dataset Q_1
4) While there is element in Q_i
5) For each dataset Q_i
6) For each point in dataset Q_i
7) Pick p ∈ Q_i : p is the first element
8) Find d(p, Q_i)
9) Split dataset Q_i in far and near set according to d(p, Q_i)
10) If Q_i is empty then remove Q_i
11) If far set is not empty then
12) Insert far set into Q
13) Set parent of first point in far set, the parent of Qi

The complete and detailed algorithm for tree construction is shown in algorithm 2, and it works as follows

First of all, a linked list is created in which each node contains the indexes of the points in data set. Now the data set and the linked list are copied to the global memory of device and the distance for each point from the first point is calculated in order to determine the level at which the root has to be placed. Since the data sets in such applications are generally too large and dataset has already been copied on device memory therefore we are using another kernel to find the distance of complete data set from the first point, which is shown in algorithm 3. The first point is inserted as root in the tree and an additional variable is kept in the data structure Queue, namely 'supernode', which keeps track of the parent node for the list being processed. The steps from (6) itself are being processed in GPU kernel. The proposed algorithm needs to synchronize all the threads at some point (10, 12) in order to make the algorithm work properly and to avoid any possibility of race conditions. Now the algorithm maintains the Queue for each level and each member of Queue is processed by different threads. As the algorithm goes executing the loop (11), the number of parallel running threads increases and the algorithm gets the benefit of parallel processing power of GPU hardware. Now, in each iteration of loop at line (11) the distance of head node from rest of all the elements are calculated if not done previously. If the maximum distance is zero then the data contains only identical points and in this case all points are inserted as children of super node. In other case, to maintain the third property of cover tree (separation), the list is divided into two lists namely the far list and the near list. The far list contains those elements which has distance greater than 2i, where "i" is the current level number. The near list contains the elements having distance less than 2i, and hence in order to maintain the property now these elements will be inserted at lower levels as children of the first element. The first element itself is also inserted as a child to maintain the covering property of tree.

There are two critical sections in the code at line (19) and (24), when updating the Queue. We need to make sure that only one thread executes the code in critical section at a time. When implementing the code, we have used the Euclidean distance to find out closeness of points in dataset.

Algorithm 2: Graphics Cover Tree Construction (GCTC) Algorithm

1) Create a linked list of data-set
2) Copy data-set and linked list on to GPU global memory
3) Find distance of first point with all points in data set and find maximum distance
4) Calculate the top level to place the root of tree
5) Create a list Queue of lists to be processed at any level
6) Find thread ID (tid)
7) If tid = 1 then
8)   Make the first element the root
9)   Make the root, the super node for the first member of list Queue
10) Synchronize all threads
11) **While** list Queue is not empty
12)   Synchronize all threads
13)   Calculate tid
14)   **While** tid < elements[Queue]
15)   **If** head node of list in Queue[tid] ≠ super node for this list
16)   Calculate distance of head node of list in Queue[tid] with rest all elements in list
17)   **If** Maximum distance is zero then
18)   Make all elements in list, the children of super node for this member of Queue
19)   Remove this list member from Queue
20)   Else
21)   Make first element of list, the child of super node for this list
22)   Split the list in far list and near list for this level
23)   Make the super Node for this list the super node for both the lists
24)   Insert the far list in Queue
25)   Tid = tid + Total number of threads
26) Copy the tree from device memory to host memory

Algorithm 3: Finding the Distance

1) Find thread ID (tid)
2) Len = Dimension of a point
3) **While** tid < Total number of points
4)   Find c = address of point[tid]
5)   Sum = 0 , j = 0
6)   **While** j < Len
7)   d1 = point [j] – c [j]
8)   d1 *= d1
9)   Sum = Sum + d1
10)   j = j + 1
11)   tid = tid + total number of threads

5. RESULTS

We measured the relative performance of our algorithm of cover tree code by comparing the execution time of our GPU version of cover tree code and it's single core version. The test machine has 8 intel Xeon processors, each working on 2.0 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 fedora11. The two platforms have been taken different intentionally to show the performance difference in traditional single core machine and cuda GPU.

The dataset for performance measurement was taken from UCI ML repository [12], and the size of dataset varies from few hundred KBs to hundred of MBs. Block size has been kept same in all cases which is 250 and 20 such blocks have been used. Table 1 shows the results and performance improvement. From these results, our algorithm is getting approximately up to three time better performance than the existing algorithm for single core CPU. Fig 4 shows the graphical representation of results.

Table 1. Results

File Name	File size (MB)	Dimensions of point	Number of points in file	Tree construction time for serial code (sec)	Tree construction time for proposed Algorithm (sec)	Performance Improvement (%)
Phy_train.data	48.8	79	50000	1.1313	0.4259	62.35
Phy_test.data	97.3	78	100000	2.3807	0.8112	65.92
Covtype.data	71.1	55	581012	6.3826	1.6718	73.80
Bio_train.data	65	76	135908	7.9179	2.4421	69.16
Bio_test.data	62	74	139658	8.4430	2.4890	70.51

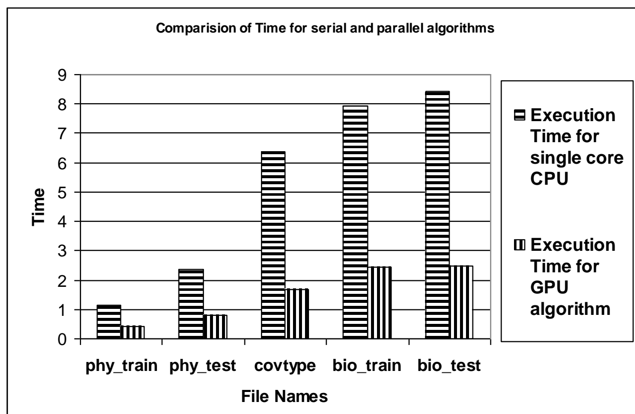

Figure 4 Results

6. REFERENCES

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

[2] Karger, D., Ruhl, M. 2002. Finding Nearest Neighbors in Growth Restricted Metrics. In Proceedings of 34th annual ACM symposium on Theory of computing, pp. 74—750.

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

[4] Clarkson, K. 1999. Nearest neighbor queries in metric spaces, Discrete and Computational Geometry. In Proceedings of 4th annual ACM-SIAM Symposium on Discrete algorithms, pp. 63–93.

[5] Krauthgamer, R., Lee, J. 2004. Navigating Nets: Simple Algorithms for Proximity Search. In Proceedings of 15th Annual Symposium on Discrete Algorithms (SODA), pp. 791-801.

[6] Beygelzimer, A., Kakade, S., Langford, J. 2006. Cover Trees for Nearest Neighbor. In Proceedings of 23rd international conference on Machine learning. pp. 97—104.

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

[8] Zhou, K., Hou, Q., Wang, R., Guo, B. 2008. Real-Time KD-Tree Construction on Graphics Hardware. ACM Transactions on Graphics. Vol. 27, article 127.

[9] Schatz, M., Trapnell, C., Delcher, A. L., Varshney, A. 2007. High-throughput sequence alignment using Graphics Processing Units. BMC bioinformatics. Vol. 8.

[10] Wu, D., Zhang, F., Ao, N., Wang, F., Liu, X., Wang, G. 2009. A Batched GPU Algorithm for Set Intersection. In Proceedings of 10th International Symposium on Pervasive Systems, Algorithms, and Networks, pp. 752—756.

[11] Kollar, T. 2006 Fast Nearest Neighbors. Technical report. Massachusetts Institute of Technology.

[12] UCI Machine Learning Repository, http://archive.ics.uci.edu/ml