GPU Accelerated Range Trees with Applications

Manoj Maramreddy and Kishore Kothapalli

International Institute of Information Technology, Hyderabad,
Gachibowli, Hyderabad, India, 500 032.

Abstract. Range searching is a primal problem in computational geometry with applications to database systems, mobile computing, geographical information systems, and the like. Defined simply, the problem is to preprocess a given a set of points in a d-dimensional space so that the points that lie inside an orthogonal query rectangle can be efficiently reported.

Many practical applications of range trees require one to process a massive amount of points and a massive number of queries. In this context, we propose an efficient parallel implementation of range trees on many-core architectures such as GPUs. We extend our implementation to query processing. While queries can be batched together to exploit inter-query parallelism, we also utilize intra-query parallelism. This inter- and intra-query parallelism greatly reduces the per query latency thereby increasing the throughput. On an input of 1 M points in a 2-dimensional space, our implementation on a single Nvidia GTX 580 GPU for constructing a range tree shows an improvement of 22X over a sequential CPU implementation. We also achieve an average throughput of 10 M queries per second for answering 4 M queries on a range tree containing 1 M points on a Nvidia GTX 580 GPU. We extend our implementation to an application where we seek to report the set of maximal points in a given orthogonal query rectangle.

1 Introduction

Manycore accelerators such as GPUs have occupied a prominent place in the theory and practice of parallel computing. This is aided in part by their ubiquitous nature, low cost, and importantly compute power. Several programming models and utility libraries such as CUDA [12], Thrust (See http://thrust.github.io/), and OpenAcc (See http://www.openacc-standard.org/) are being currently supported for writing general purpose programs on GPUs. It is possible to arrive at very efficient implementations of general purpose computations using such programming support [11, 2].

On the other hand, there is very little work on how to efficiently build and operate on data structures on architectures such as GPUs. Hierarchical data structures such as trees and multi-dimensional data structures render the nature of the problem more difficult. In fact, there are very few such reported instances in the literature. Some early work in this direction by Lefohn et al. [10] proposes a template library that can be used to build data structures and also identify common themes across existing GPU based data structres. It is to be noted that very few of the existing data structures deal with hierarchical ones [3, 6].

We posit that it is possible to build hierarchical data structures on GPUs by introducing novel techniques that improve the way the data structures are built and accessed. Given the plethora of programming support available for programming GPUs, we show that it is also possible to build such data structures with a minimal programming effort. Our work indicates that one should make use of available primitives such as `sort`, `merge`, and `scan`.

As a case-study, we consider data structures for multi-dimensional datasets such as the range tree. A d-dimensional range tree is a data structure that can store a set of points in a d-dimensional space so that operations such as searching on d-dimensional datasets is efficiently supported. A d-dimensional range tree for storing n points requires a space of $O(n \log^{d-1} n)$ and involves creating a nested set of d trees. In a sequential setting, this construction can be done efficiently in time $O(n \log^{d-1} n)$, and the data structure can be stored using pointers. However, creating and accessing pointer based data structures is difficult in many-threaded settings. Hence, one has to identify alternate ways to represent the data structure while keeping access to the data structure as efficient as possible. Fortunately, the nested trees that arise in the range tree are all full binary trees, i.e., for n nodes where n is a power of 2, these trees have exactly $\log n$ levels. We use this fact, along with additional novel considerations to represent a d-dimensional range tree. For building a 2-dimensional range tree on an input dataset of 2^{20} points we achieve a speed up of 22X compared to the sequential CPU implementation.

We also show that accessing the range tree in our representation is also efficient by considering two canonical applications. One of the prominent applications of range trees is range searching. In range searching, one is interested in reporting the points that lie in a given orthogonal query rectangle. An orthogonal query rectangle is a rectangle whose sides are parallel to the axes of the d-dimensional space. Such a query finds applications in several areas such as database systems, geographical information systems, mobile computing, CAD tools and the like [1]. In this case, it is easy to notice that there is natural inter-query parallelism. However, we also modify the querying algorithm to exploit also intra-query parallelism. The combination of the two help us in increasing the query throughput.

The second application we consider is that of reporting the set of maximal points that lie in a given orthogonal query rectangle. A point $P = (x_1, x_2, \cdots, x_d)$ is said to be *maximal* if no point $P' = (x'_1, x'_2, \cdots x'_d)$ exists with $x_i < x'_i$ for $1 \leq i \leq d$. The set of maximal points, also called as *skyline points* offer a good summarization of the points. For this problem also, we exploit intra-query parallelism and by introducing a standard primitive called the All-Nearesest-Larger-Values (`ANLV`) [7]. This primitive that we develop as part of this work can be of independent interest to the parallel computing community.

For both of our applications, on trees with 2^{10} points and 1 M queries, our implementation on an NVidia GTX 580 GPU achieves around 7x more throughput compared to a 12-threaded implementation on an Intel i7 X980 CPU.

1.1 Related Work

Efficient constructions of hierarchical data structures on modern architectures is an emerging research theme. Construction of B+ trees is studied in [5], and of KD-trees in [6]. Kim et al [9] had proposed solutions for implementing R-trees on GPUs. They propose solutions to avoid irregular memory access and improved efficiency. A Massively Parallel Three-phase Scanning (MPTS) algorithm for R-tree traversal for processing multi-dimensional range queries is proposed in [9]. Both of the works focuses on R-tree search algorithms, but not on constructing the trees in parallel. To the best of out knowledge this is the first attempt to implement range trees on GPUs. We provide solutions for both efficient construction and accessing of range trees on GPUs.

2 Preliminaries

For ease of exposition, we describe the 2-dimensional range tree in the following. In this case, we assume that each point has a x-coordinate and a y-coordinate. In a 2-dimensional range tree, we start with a *primary tree* that is a balanced binary search tree T built on x-coordinate of points in P. Each internal node in the primary tree can be the median of the canonical subset of v. Further, every node v in T contains a pointer to a *secondary tree* that is a binary search tree on y-coordinate of the canonical subset of v. The space required for a 2-dimensional range tree is $O(n \log n)$. An example is shown in Figure 1. The time required for constructing a 2-dimensional range tree is $O(n \log n)$. Points stored in the leaves of a subtree rooted at an internal node v are called the *canonical subset* of v and v is called the canonical node of the subset.

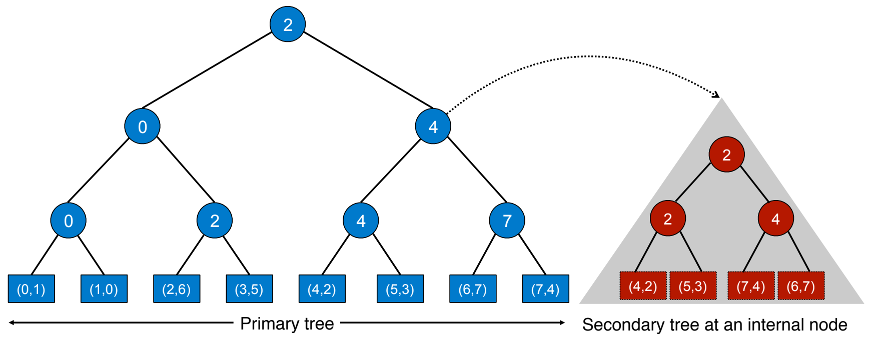


Fig. 1. A 2-dimensional range tree. Each node in the primary tree has a corresponding secondary tree.

2.1 Range Querying

We first describe a 1-dimensional range query. To report the points in a 1-dimensional query $[a, b]$ we proceed as follows. We first search for a and b in the 1-dimensional range tree. Let u and v be the leaves where this search ends. The points in the interval $[a, b]$ are the points stored in the leaves in between u and v and possibly the points stored at u and v. Let r_{split} be the node where the search paths to u and v separate. Starting from r_{split} we then follow the search path to u. At each node where the path goes left we report all the leaves in the right subtree. Similarly, we follow the path to v and report all the points in the left subtree when the path goes right.

For a 2-dimensional query $[a, b] \times [c, d]$, we first perform a 1-dimensional query $[a, b]$ on primary tree. At each node where the search path to u goes left we do a 1-dimensional range query on y-coordinate in the secondary tree associated at the right child of the node. Similarly, at each node where the search path to v goes right we do a 1-dimensional range query on y-coordinate in the secondary tree associated at the left child of the node.

3 A Parallel Range Tree

In this section we describe our new implementation of a multi-dimensional range tree. We argue that our new representation is efficient on GPU architectures in the way that it is represented and accessed in parallel. In a standard two-dimensional range tree (cf. Section 2), every node in the primary and secondary tree contains left and right child pointers. Porting this representation of range tree directly on to the GPU platform is not efficient due to the below mentioned reasons.

- Accessing multiple levels of pointer indirection will lead to massively increased memory access latency and will break the little cache coherency available on the GPU.
- The irregular tree traversals cause thread divergence when implemented on GPUs. Also, one needs to regularize the work done by each thread in order to achieve maximum efficiency.
- Copying a complex structure such as range trees, consisting of nested pointers, on to GPU requires a *deep copy* functionality for which there is no available API. Copying back the same structure poses the same problem in reverse.

For addressing the above mentioned challenges, we use an array based representation of complete range tree. We flatten-out the hierarchical structure of the range tree into a structure containing two 1-dimensional arrays, one storing x-coordinates and other storing y-coordinates. We first store the primary tree in an array, followed by the secondary trees from bottom up approach. We label the nodes in the tree in inorder starting from '0'. Rather than storing inorder traversal of the complete tree, we only store the leaves of primary and secondary trees. By using bitwise representation of the nodes it is possible to dynamically

converting array index i to corresponding index j of the leaf in primary tree	$j = 2 \times i + 1$
converting internal node(range tree) index i to array index j	$j = i/2 - 1$
computing offset of secondary tree at node i of primary tree	$h = \log_2(i\& - i)$ $\text{offset} = h \times n + i \times 2^{(h+1)} \times (i\& - i)$

Table 1. Conversion formulas between array representation and standard range tree.

compute the offset of secondary trees and corresponding internal nodes of the trees. Formula for converting indices from array representation to virtual range tree is given in table 1. An example of this representation is given in Figure 2. The point set used is same as in Figure 1.

This simple structure also helps in building the range tree using existing primitives such as sorting and merging. The representation not only helps us in regularizing the work done by threads while processing the queries but also avoids the increased memory access latency that might arise due to multiple levels of pointer indirection. We use the bit representation to exploit intra-query parallelism as explained in Section 4. Further, our representation requires the same space asymptotically as the standard representation. Finally, though we perform our experiments in a two dimensional space, the same representation can be extended to higher dimensions.

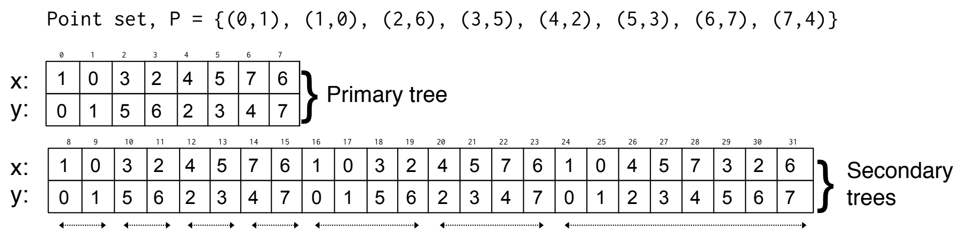

Point set, P = {(0,1), (1,0), (2,6), (3,5), (4,2), (5,3), (6,7), (7,4)}
0 1 2 3 4 5 6 7
x: 1 0 3 2 4 5 7 6
y: 0 1 5 6 2 3 4 7
Primary tree
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
x: 1 0 3 2 4 5 7 6 1 0 3 2 4 5 7 6 1 0 4 5 7 3 2 6
y: 0 1 5 6 2 3 4 7 0 1 5 6 2 3 4 7 0 1 2 3 4 5 6 7
Secondary trees

Fig. 2. In the new representation only leaves of each tree (primary & secondary) are stored in an array. The trees are stored contiguously. The offset of secondary trees and the corresponding internal nodes can be dynamically computed.

3.1 Implementation details

Our array based representation of range trees helps us in using existing primitives for construction. In the following, we show the steps for constructing a 2-dimensional range tree.

1. Sort the points on their x-coordinates. Sorting is a well studied problem on GPUs. For this purpose, we use `sort_by_key` implementation provided by Thrust library.
2. Merge the points recursively on their y-coordinates and store the merged result in each iteration. The merged result at each iteration in fact represents secondary trees from bottom to top. Though `thrust` provides a merge implementation, we use a more recent merge sort implementation (`kernelMerge`) provided by Baxter (See http://nvlabs.github.io/moderngpu/).

Algorithm 1 BUILDRANGETREE(P)

```
1: Input. Set of points in a two dimensional plane $P := \{p_1, p_2, ..., p_n\}$.
2: Output. A 2d Range tree $T$.
3: $T[1...n] \leftarrow$ sort_by_key($P$)
4: $numPasses \leftarrow log(n)$
5: for $pass = 0, numPasses$ do
6:     $coop = 2^{(pass+1)}$
7:     $source = T[n \times pass...n \times (pass + 1)]$
8:     $dest = T[n \times (pass + 1) + 1...n \times (pass + 1) + n]$
9:     kernelMerge(source, dest, coop)
```

3.2 Results and Performance Analysis

Platform: All our experiments are performed on a machine with Intel core i7 X980 CPU and Nvidia Geforce GTX 580 GPU. The Intel core i7 X980 is a 3.33-GHz six-core CPU with Intel's hyper-threading technology. It can work on 12 streams at once. It has a 12MB L3 cache. GTX 580 has 512 cuda cores. For all our experiments we used OpenMP specification 3.0 and the CUDA 5.0 programming model for programming multi-core CPUs and Nvidia many-core GPUs respectively.

Dataset: For input data we have randomly generated points from a uniform distribution. We perform our experiments on data sets with small trees containing few thousand points to large trees with over 1 M points.

Our simplified structure of range tree enables us to achieve faster construction times using existing primitives such as sorting and merging. With minimal programing effort we are able to achieve faster construction times. Figure 3 shows the speed up of constructing a range tree on GPU over a sequential CPU implementation. It is evident from the graph that our implementation can easily scale to huge datasets. For constructing a range tree on a dataset with 1 M points, we achieve a speed up of 22X on GPU over sequential CPU implementation.

4 Application I: Range Searching

The problem of range searching is to report the set of points that lie in a given orthogonal query rectangle. An orthogonal query rectangle has its sides paral-

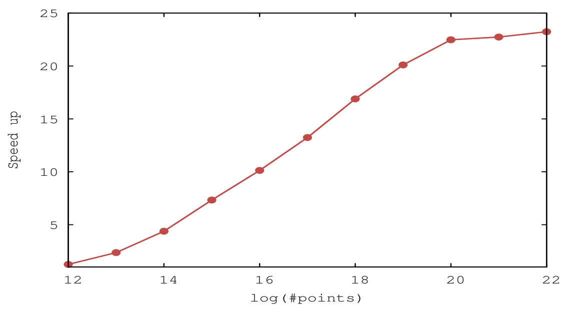


Fig. 3. Speedup of building range tree on GPU vs sequential CPU implementation.

lel to the axes of the underlying space. An orthogonal rectangle can then be represented by considering the cross product of ranges in each dimension. In particular, in a 2-dimensional setting, the rectangle $[a, b] \times [c, d]$ refers to the rectangle consisting of points whose x-coordinates are in $[a, b]$ and y-coordinates are in $[c, d]$. Given a range tree for n points in a 2-dimensional space and a range query q of the form $[a, b] \times [c, d]$, the algorithm to process the query has the following three main steps (cf. Section 2).

1. Finds the nodes that are closest to a and b in the primary tree
2. Find the canonical nodes in the primary tree, and
3. Find the result by repeating the above steps in the secondary tree for each canonical node of the primary tree
4. Transfer results to host CPU

In the following, we show how each of the above steps can be also performed in parallel for a given query. This helps us extract intra-query parallelism apart from the standard inter-query parallelism.

1. **Binary Search on Primary tree:** In the first phase, we binary search for a and b in the primary tree. We assign a search key per thread. In order to avoid conditional branching of threads we store our sorted array in level-order rather than in-order. This technique was used in [11] to avoid conditional branching of threads.
   Let u and v are the indices of the nodes where the binary search for a and b in the primary tree ends. The split node r_{split} is computed by taking `xor` of u and v. The number of canonical nodes can be obtained by counting the number of set bits. The result of the binary search and the split nodes are saved and passed as input to Phase-2.
2. **Find canonical nodes:** In the second phase, we compute the indices of the canonical nodes in parallel. While the standard range search implementation is bounded by sequential search for canonical nodes, we present a method to find all the canonical nodes in parallel. In order to get the total number of canonical nodes for the batch of queries we perform a parallel reduce on the number of canonical nodes for each query obtained from Phase-I.

Algorithm 2 TRAVERSETREE(key)

```
1: $j = 1$
2: **repeat** $log\ k$ times
3:     $j = 2j + (key > bst[j])$
```

Algorithm 3 PHASE-I: KernelPrimaryTreeSearch

```
1: *Input.* A 2-dimensional range tree $T$ and a batch of orthogonal range queries $Q$,
   $q_i = [a : b] \times [c : d]$.
2: *Output.* Index of leaf where the binary search ends $\forall$ x-coordinate of queries, $q_i \in Q$.
3: *#threads.* A batch of $2 * |Q|$ threads are launched, where each thread searches for
   a key in the binary search tree.
4: **for** $i = 0$ to $(2 * |Q|)$ **do**
5:     $key = queries\_x[i]$
6:     $j =$TRAVERSETREE($key$)
7:     Check if the node at $j$ lies in the range.
8:     $j = 2 * j + 1$ // converting into virtual rangetree index
9:     $bs\_output[i] = j$
10:    __syncthreads()
11:    $k =$fetch the right bs_output if left and vice versa
12:    $t = FindLog2(k \oplus t)$
13:    **if** (thread index is odd) **then**
14:        $k = k \ \& \ (2^t - 2)$
15:    **else**
16:        $k = \sim k \ \& \ (2^t - 2)$
17:    counting set bits in $k$ gives number of canonical nodes
```

The inorder labeling of the nodes in the tree provides information about the path traced from root to that node. A $'0'$ bit at i^{th} position from right indicates the path has traversed left and a $'1'$ bit indicates the path has traversed right. Using this path information and the corresponding split node obtained in Phase-I, we give a technique to compute the canonical nodes in parallel.

In the path from split node to u, a $'0'$ bit indicates the presence of a canonical node. Similarly, for the right path to v a $'1'$ bit indicates the presence of a canonical node.

3. **Binary Search on Secondary tree:** For every canonical node found in Phase-2, we perform binary search for c and d in the corresponding secondary tree. The output of the binary search for each canonical node is stored.

Algorithm 4 PHASE-II: FindCanonicalNodes

```
1: Input. Binary search result on primary tree and split nodes.
2: Output. Canonical nodes of all queries in primary search tree.
3: #threads. Number of canonical nodes.
4: $q_i = query\_index$
5: $l = bs\_output[q_i]$
6: // for computing $i^{th}$ canonical node of $q_i$
7: $k$ = find the position of set bit in $l$
8: clear last $k$ bits in $l$
9: canonical_node = $l + 2^{(k-1)}$
```

Algorithm 5 PHASE-III: SecondaryTreeSearch

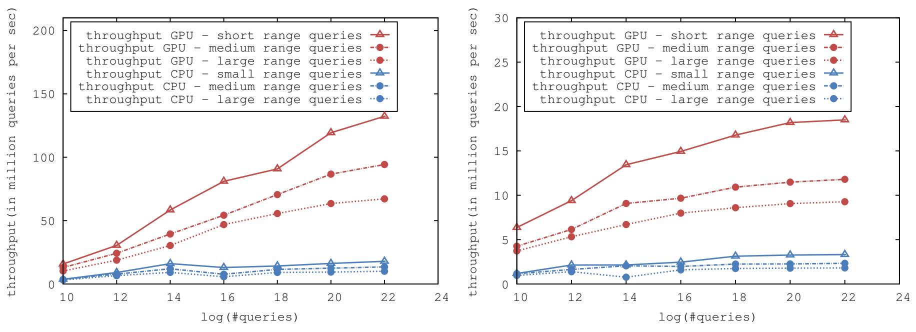
```
1: Input. Canonical nodes of all queries in primary search tree and batch queries $Q$,
   $q_i = [a : b] \times [c : d]$.
2: Output. Output of binary search in secondary trees $\forall$ y-coordinate of queries, $q_i \in$
   $Q$.
3: #threads. A batch of $2*|can\_node|$ threads are launched, where each thread searches
   for a key in the binary search tree.
4: for $i = 0$ to $(2 * |can\_node|)$ do
5:     $key = queries\_y[i]$
6:     $j$ =TRAVERSETREE($key$)
7:     Check if the node at $j$ lies in the range.
8:     $Output[i] = j$
```

4. **Reporting results:** The number of output points generated per query can be of the order of $O(n)$. Copying back such huge data to the host CPU consumes a significant amount of time. We alleviate this problem substantially by reporting only the left and right indices of our search in secondary trees. A sequential scan of these ranges on the host would output the points on the host side. This greatly reduces the amount of data to be transfered to $O(\log n)$ per query. In order to further hide the copy time we process our queries in batches so that the copy time of output of the i^{th} batch can be completely hidden by computation of the $(i-1)^{th}$ batch.

4.1 Performance Analysis

Dataset: To generate the queries, we study three different datasets. These datasets are dictated by the number of canonical nodes that each query results in. Since the number of canonical nodes in each query directly impacts the work done in Phase II and III of our querying algorithm, this study helps us understand the efficacy of our implementation. It is easy to note that in a range tree containing n points, the average number of canonical nodes in a query whose range is generated uniformly at random is $O(\log n/2)$. Based on this average, we study the following query datasets.

1. **Short-range Queries:** We define a query as a short-range query if the number of canonical nodes for the query is between zero and $\log n/2$.
2. **Medium-range Queries:** A medium-ranged query has canonical nodes between $\log n/2$ and $3 \log n/4$.
3. **Long-range Queries:** Any query with canonical nodes greater than $3 \log(n)/4$ is defined as a long-range query.



(a) Throughput graph on Small trees (b) Throughput graph on Large trees

Fig. 4. Throughput of GPU range searching vs. a 6-core CPU

Throughput: In our experiments, we consider trees with 2^{10} points as small trees and trees with 2^{20} points as large trees. The throughput graph for the three datasets is show in Figure 4. We see from Figure 4 that our algorithm scales for both small trees and large trees and also over the three query datasets. This suggests that a batch of queries that come with a mix of short-range to long range queries can also be processed without any further rearrangement of the queries. We do notice a higher throughput for the Short-range Query dataset compared to other datasets. This is due to the fact that as the number of canonical nodes is small in that dataset, the amount of computation spent in phase 2 and 3 is minimal.

Batch Size: We finally study the impact of batch size on our implementation. Recall from Section 4, Phase IV of our querying algorithm can be made to overlap with Phases I-III of the querying algorithm a scenario, finding the right value for the batch size is crucial. In Figure 5 we show the throughput achieved by our algorithm as a function of the batch size. As can be intuitively observed, the throughput increases with increasing batch size up to a certain point. This is due to the fact that the time spent in Phase IV can be completely hidden by the time spent in Phase I-III, except for the time spent in Phase IV for the last batch. However, as we increase the batch size further, the increase in time spent in Phase IV will decrease the throughput achieved. From Figure 5, we notice that for a dataset of 2^{16} points and 2^{16} queries, a batch size of 2^{13} is ideal.

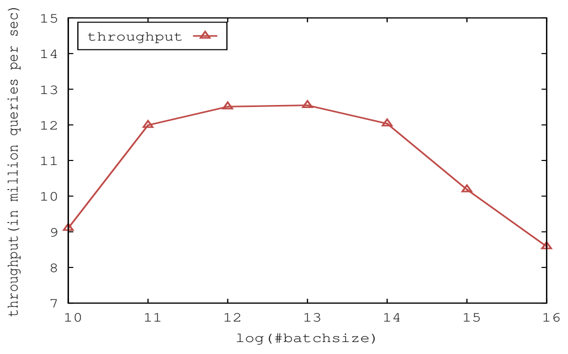
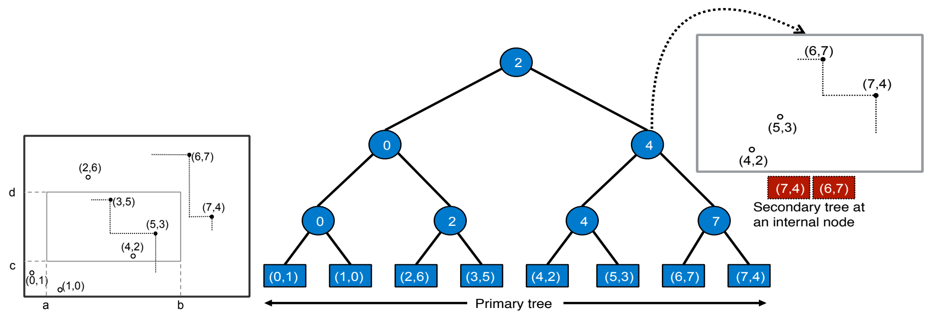


Fig. 5. Variation of query throughput with batch size for a dataset of 2^{16} points and 2^{16} queries.

5 Application II: Reporting Maximal Points in an Orthogonal Query

A traditional range search query focuses on returning all the points inside a given range. But when dealing with large datasets, the resulting number of points may be huge and hence it is impractical to return the entire result. One such scenario may be server returning results to mobile devices where bandwidth and screen resolution are constrained. In such scenarios, it is beneficial to return a summary of the result. Maximal points offer a good summary of the results [1]. Using range trees, a sequential algorithm to report the set of maximal points in a given orthogonal query region is proposed in [4]. In this section, we use our GPU-based construction of a range tree to provide a parallel solution to problem of reporting the maximal points in a given orthogonal query region.



(a) Skyline points inside a rectangle

(b) In the associated structure we only store the skyline of the points rooted at that internal node

Fig. 6. An application of range trees to find maximal points inside a query rectangle.

In order to efficiently report the maximal points, also called as the *skyline points* , or simply the *skyline*, inside a given range, we preprocess the input point set into a data structure. This data structure is similar to range tree we described in Section 3 with a difference that we store the maximal points of the canonical subset of points at each internal node of the secondary tree. This solution is described in detail in [8].

The algorithm for reporting skyline points inside an orthogonal range query is similar to that of range searching described in Section 4. At the end of Phase-3, we get a skyline corresponding to each canonical node in the primary tree. Merging these canonical-skylines produces the final skyline inside the query range. Below we explain the steps required to merge the skylines.

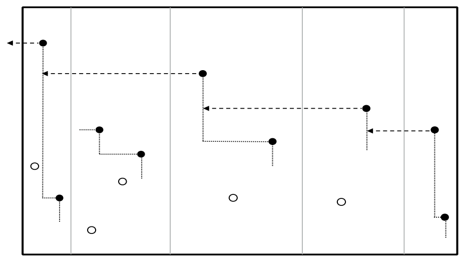
1. **Filtering overshadowed skylines:** A skyline S_i is said to be overshadowed by skyline S_j if the maximum y-coordinate of points in S_j is greater than maximum y-coordinate of points in S_i. All canonical-skylines may not contribute to the final merged output as they may be overshadowed by a skyline to their right. We filter such skylines prior to merging them as follows.
   Let $Y_{max} = \{y_1, y_2, ...y_k\}$ be maximum y-coordinates of the points in each of the canonical-skylines. For filtering the skylines we perform an All Nearest Larger Value (`ANLV`) to the left on Y_{max}. The problem of `ANLV` is defined as follows. Given an array A of n elements, for each element $A[i]$, find the element closest to the left of i that is greater than $A[i]$. `ANLV` is a well studied problem in parallel algorithms [7]. For solving this problem, we use the algorithm from [7]. For every canonical-skyline we find a target skyline to be merged with. An example of the problem is illustrated in Figure 7.
2. **Merging skylines:** Assuming S_i and S_j are the two skylines to be merged where S_i lies left of S_j. Let y_i be the maximum y-coordinates of the points in S_j. In order to merge S_i and S_j we find the merge point by searching for y_i in S_i. For all the canonical-skylines that are not filtered in the above step we find the merge point by searching their maximum y-coordinate in the corresponding target skyline.
3. **Reporting results:** By traversing the skylines from right to left through the merge points, we can obtain the final skyline. But since this operation is sequential in nature, we perform the traversal on the host CPU. The set of canonical-skylines that are not filtered and their corresponding merge points are returned as a result set from GPU to host CPU.

5.1 Performance Results

For the experiments, we generate random queries from a uniform distribution. The throughput graph is show in Figure 8. As can be seen, our implementation offers a good speed-up over a corresponding multi-core CPU.

6 Conclusions

In this paper, we show that hierarchical data strucutres can be efficiently constructed on modern parallel architectures. Our method involves identifying efficient ways to store and represent the data structure without compromising on


(a) We compute ANLV on the maximum y-coordinates of points in skylines

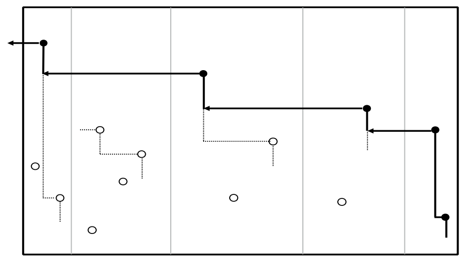
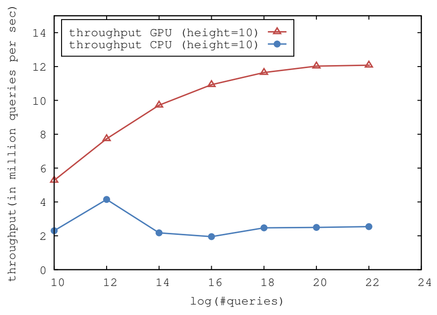
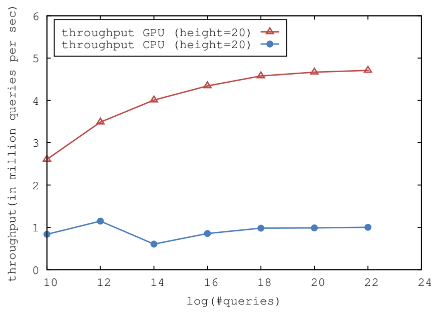

(b) Final skyline is obtained by merging individual skylines

Fig. 7. Finding skyline inside an orthogonal range query



(a) Small trees



(b) Large trees

Fig. 8. Throughput graph of GPU vs 6-core CPU

the access efficiency of the represenation. As a case-study, we considered the range tree along with two applications of the same.

References

1. P. K. Agarwal and J. Erickson. *Geometric range searching and its relatives.* In Advances in Discrete and Computational Geometry, volume 223, pp. 1-56.
2. N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Technical Report NVR-2008-004, 2008
3. G. Coombe and M. J. Harris and A. Lastra. Radiosity on graphics hardware. In Proceedings of the 2004 Conference on Graphics Interface. 161–168, 2004.
4. Ananda Swarup Das, Prosenjit Gupta, Kannan Srinathan *On Finding Skyline Points for Range Queries in Plane* CCCG 2011
5. Jordan Fix, Andrew Wilkes, Kevin Skadron *Accelerating Braided B+ Tree Searches on a GPU with CUDA* In Proc. ISCA Workshops, 2011.
6. Foley, T., and Sugerman, J. *Kd-tree acceleration structures for a gpu raytracer* In Proc. Graphics hardware, pp: 15–22, 2005.
7. Joseph Jaja, An Introduction To Parallel Algorithms, 2004, Addison-Wesley.

8. A. Kalavagattu, J. Agarwal, A. Das, and K. Kothapalli. On Counting Range Maxima Points in Plane, in Proc. IWOCA 2012, pp. 263-273.
9. Jinwoong Kim, Sul-Gi Kim, Beomseok Nam *Parallel multi-dimensional range query processing with R-trees on GPU* J. Par. Dist. Comp., 73(8), 2013, 1195-1207.
10. A. E. Lefohn and S. Sengupta and J. Kniss and R. Strzodka and J. D. Owens. Glift: Generic, Efficient, Random-access GPU Data Structures, ACM Trans. Graph., 25(1), 2006, pp: 60–99.
11. N. Leischner, V. Osipov, and P. Sanders. *GPU sample sort.* IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
12. NVidia Corporation, "Cuda: Compute Unified Device Architecture programming guide," Technical report, Nvidia, Tech. Rep., 2007.