Spatial Indexing of Large-Scale Geo-Referenced Point Data on GPGPUs Using Parallel Primitives

Jianting Zhang  
Department of Computer Science  
The City College of the City University of New York  /New York, NY, 10031  
jzhang@cs.ccny.cuny.edu

Le Gruenwald  
School of Computer Science  
University of Oklahoma  /Norman, OK 73071  
ggruenwald@ou.edu

ABSTRACT
Modern positioning and locating technologies, e.g., GPS, have generated huge amounts of geo-referenced point data that are crucial to understand environmental and social-economic phenomena. Unfortunately, traditional disk-resident databases are inefficient in handling large-scale point data. In this study, we propose to utilize the massive data parallel processing power of General Purpose computing on Graphics Processing Units (GPGPUs) technologies to index large-scale geo-referenced point data by using parallel primitives for efficiency, simplicity and portability. We have developed a CSPT-P (Constrained Space Partitioning tree for Point data) tree indexing structure that is suitable for parallel construction. Experiment results using a New York City (NYC) taxi trip dataset with nearly 170 million taxi pickup locations have demonstrated a 23X speedup on an Nvidia Quadro 6000 device over a serial CPU implementation on an Intel XEON E5405 processor.

1. INTRODUCTION
Spatial indexing is widely used in spatial databases, computer graphics, information retrieval and Geographical Information System (GIS). Over the past decades, numerous spatial indexing techniques have been developed [1, 2]. While parallel spatial indexing has attracted considerable research interests, existing works on parallelization of spatial index construction for query processing in the context of geospatial data management are mostly designed for distributed shared-nothing computing nodes. The recently emerging GPGPU technologies have made it possible to use for general computing. A few GPGPU based spatial indexing approaches have been proposed to speed up ray tracing and graphics rendering [3-5]. However, there is little research on utilizing the massively data parallel processing power of GPUs for geospatial data management despite its potentials (see [6] for more detailed discussions). Spatial indexing techniques that have recently been proposed for graphics applications may not be suitable for geospatial data management due to their rendering rather than query focus. Furthermore, these techniques require deep understanding of considerable hardware programming details which is usually not very attractive from a data management perspective.

A natural remedy is to utilize high level parallel programming libraries or APIs to hide hardware details while exploiting the parallel processing power of GPUs. Quite a few GPGPU based parallel libraries are currently available that provide efficient implementations of parallel primitives on GPGPUs. Parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines. The behaviors of popular parallel primitives on vector data are well-understood. Such primitives have been successfully used in many applications, including relational data management [7]. Unfortunately, to the best of our knowledge, the spatial data management community has not yet widely embraced GPGPU technologies. Indeed, traditional spatial indexing approaches heavily rely on tree structures and many of the index construction algorithms are sequential in nature. It is technically challenging to utilize massively data parallel architectures such as GPGPUs for spatial indexing. We believe that identifying the inherent parallelism in spatial indexing is crucial to the development of efficient parallel spatial indexing algorithms. Mapping such parallelism to generic parallel primitives is a first step to develop spatial-specific parallel primitives for more efficient implementations on different hardware.

Towards this end, we have developed a primitive-based spatial indexing approach for large-scale point data. Our design adopts a level-wise processing schema for Z-ordered [9] points and uses parallel primitives, such as transform, sort, scan, and reduce, extensively to process data items. Additional scatter and gather primitives are used to build the correspondences between parent nodes and child nodes and construct tree structures in parallel. While we currently focus on constructing Constrained Space Partition Trees for Point data (or CSPT-P trees), we believe our proposed approach can be generalized and applied to various hierarchical spatial indexing structures. The resulting CSPT-P trees can be directly used for query processing in spatial databases and load balancing in geospatial computing. Our preliminary implementation using the Thrust library [10] have demonstrated a 23X speedup on an Nvidia Quadro 6000 card over a serial implementation on an Intel E5405 CPU using just a few hundreds lines of code. Our preliminary results are encouraging in the sense that it is possible to perform multidimensional spatial indexing and geospatial data management on GPGPUs in parallel with reasonable coding and development efforts. The high cost-effectiveness ratio may stimulate further research, development and application interests in the broad geospatial data management community.

The rest of the paper is arranged as follows. Section 2 briefly introduces the background and related work. Section 3 presents the Constrained Space Partition Tree for Point data (CSPT-P) structure to index large-scale point datasets. Section 4 provides the tree construction algorithm and implementation details. Section 5 describes the experiment results on a real dataset consisting of approximately 170 million taxicab pickup locations in 2009 in NYC. Finally Section 6 is the conclusions and future work directions.

2. BACKGROUND, MOTIVATIONS AND RELATED WORKS
Point data are universal and many techniques exist to manage and process them. In the context of geospatial data management in spatial databases, GIS and geospatial modeling, point data are usually locations on the Earth surface which can be represented as (longitude, latitude) pairs. Geo-referenced points often are associated with a set of attributes, such as elevation, precipitation, bus stop identifier or business name. A variety of analysis can be performed on the locations, attributes or their combinations. In the Simple Feature Specification (SFS) [11] defined by the Open Geospatial Consortium (OGC) and adopted by the International Standard Organization (ISO), points are the basis to construct more complex geospatial features, such
as polylines and polygons. Modern positioning technologies, such as the Global Positioning System (GPS), have generated tremendous geo-referenced points and they may be crucial for us to understand a variety of natural and social-economic phenomena. For example, there are nearly half a million taxi trip records in the New York City (NYC) per day and each trip has a pick-up and drop-off location. Traffic and travel patterns discovered from the taxi pick-up and drop-off locations can be important in facilitating urban planning and traffic mitigation. We also believe that geo-referenced cell-phone call records and geocoded addresses from social network data can be important non-traditional point data sources to understand social dynamics.

While several leading commercial and open source databases, such as Oracle, DB2 and PostgreSQL have provided geospatial data management capabilities for nearly a decade and Microsoft and MySQL also began to support geospatial data in their most recent releases, these disk-resident databases running on a single CPU are not designed to process large-scale geospatial data with desired high performance. Our experiments on managing 177 million taxi trip records show that indexing the pick-up or drop-off locations on a PostgreSQL 9.0 instance took more than 50 hours which is 3-4 orders slower than our desired performance (e.g., in a few seconds). Motivated by the practical needs of efficiently and effectively managing large-scale taxi trip records, we have decided to explore the parallel processing power of GPGPUs. As a first step, we focus on identifying “hot spots” with dense pick-up and drop-off locations. We thus wish to develop a lightweight hierarchical tree indexing structure to speed up the indexing process. We have leveraged our previous experiences in designing cache-conscious, low memory footprint and array-based pointerless quadtrees for raster data [12]. Since our interests are on clusters, quadrants whose number of points that are below a threshold K can be identified and the tree construction process at the quadrants can be stopped. The constraint has also been discussed in some other application contexts [13]. In addition to the practical concern, we also believe that it is methodologically more interesting to develop a tree index construction algorithm directly on points (vs. raster) that are analogous to spare matrix (vs. dense matrix). We would like to stress that, while the techniques we presented in this paper are motivated by a specific application on processing large-scale taxi trip records, we believe they are generic enough to be applied to other types of large-scale point data.

GPGPU technologies have attracted quite a lot of interests in research and applications from many areas including databases. A set of GPGPU primitives to support relational operators have been developed on top of classic parallel algorithms including sorting and scan [7]. Bakum and Skadron have implemented a subset of the SQLite command processor directly on GPU [8]. While these pioneering works have demonstrated the effectiveness of speeding up read-only relational operations and have set a solid base for further investigations, it is unclear how they can be extended to indexing and querying multidimensional geospatial data as relational data and geospatial have quite different characteristics.

3 THE CSPT-P TREE INDEXING STRUCTURE

The proposed indexing structure is an extension to the popular quadtree indexing tree family [1, 2]. The extension has two enhancements. The first enhancement is to allow variable fan-outs at each level of the tree structure instead of using a fixed fan-out of 4 in quadtrees. The second enhancement is to stop tree construction at nodes with number of points that is below a certain limit. Naturally the CSPT-P indexing structure belongs to Space Partitioning Tree (SPT) [14] due to the first enhancement. The second enhancement is actually a constraint on the tree structure. As we shall see more clearly later, the constraint not only makes the tree construction more efficient, but more importantly, it represents real world constraint and can be applied to various applications, such as spatial summarization/statistics, visualization, page-oriented storage and load-balancing in parallel computing. The name of the proposed indexing structure reflects the two enhancements precisely. An example of the CSPT-P tree is shown in Fig. 1. Since the fan-out is 4, it is a quadtree in this particular example. Conceptually, the indexing space is recursively partitioned until the leaf nodes are reached. Here a leaf node represents a quadrant that has no more than K points.

The physical layout of the CSPT-P tree shown in Fig. 1 is provided in Fig. 2 which is similar to that of the CCQ-Tree that we have designed for indexing raster geospatial data on GPGPUs [12]. All tree nodes are stored in a 1-D array which makes it easy to be transferred among disk, CPU memory and GPU memory without needing serialization. In our design, each CSPT-P tree node has four fields, i.e., key, type, number of children and first-child position. Different from traditional main-memory space partitioning trees where each node has N pointers each of which points to a child node, we only keep the position of the first child to reduce memory footprint and improve cache efficiency as discussed in the designs of Cache Sensitive Search Tree (CSS-Tree) and Cache Sensitive B+-Tree (CSB+-Tree) by Rao and Ross [15, 16].

![Fig. 1 An example of CSPT-P Tree](image)

The semantics of the numbers and positions of the first child positions are designed to be different for leaf nodes and non-leaf nodes. Since child pointers in the leaf nodes of traditional space partitioning trees are NULL which is a waste of space, we re-use the space to represent the numbers of points and the starting positions of the first points in the blocks of point data array. The type field in a CSPT-P tree node tells whether the node is a leaf node (i.e., 1) or non-leaf node (i.e., 0). The number and starting position fields stored in the leaf nodes can be used to efficiently retrieve points after a query using the indexing tree. Obviously, the points among blocks are sequenced in the same order as the leaf-nodes using a Breadth-First Search (BFS) traversal order. However, points within blocks are not ordered as they are not required by our application (as well as in many other applications). Since the block capacity K can be large (e.g., 2000), the design can significantly save tree construction time due to a novel way to reduce sorting overhead as discussed in details in the next section. A similar discussion has been provided in the context of disk I/O efficient quadtree construction for generating grid Digital Elevation Model (DEM) from point clouds [17].
4 CSPT-P TREE CONSTRUCTION USING PARALLEL PRIMITIVES

Our goal is to design an approach to constructing CSPT-P trees in parallel that balances between efficiency and implementation complexity. While it is certainly possible to develop faster implementations by exploiting more hardware specific features, we are interested in utilizing high-level parallel primitives that can effectively make use of the parallel processing power of GPGPUs without having to be exposed to too many hardware details which could be a limiting factor for practical applications. Before we present the details of the construction algorithm, we provide an overview of our design. While sorting has been frequently used in parallel tree construction, instead of sorting all the points at the finest level directly and building tree structures bottom-up, we incrementally sort points based on their increasingly longer keys level by level and remove points in leaf quadrants as early as possible in a top-down manner. Since our CSPT-P tree allows up to K points in leaf quadrants (or leaf points for short), these leaf points do not need to be sorted within the respective quadrants which significantly save sorting time. Also, as the sorting keys become longer, the numbers of points to be sorted become smaller after removing such leaf points. It is easy to find out that the order of the identified leaf quadrants follows a BFS order. Based on the BFS-ordered leaf nodes, we proceed to construct a CSPT-P tree by filling up the non-leaf nodes, also in a level-wise, top-down manner. While CSPT-Tree nodes (including both leaf and non-leaf nodes) are being generated in parallel at each level by computing the numbers of children and the starting positions of the first child nodes, the information associated with the leaf nodes derived previously is combined. We divide the whole tree construction process into two parts each of which consists of multiple steps and each step corresponds to a parallel primitive. We refer to Appendix 1 for the brief descriptions of the parallel primitives that we have used in our design.

Part 1: Identifying leaf nodes level by level.

Inputs: vector of point dataset \( P \)

Outputs: re-arranged point dataset \( P \), leaf key vector \( LK \), vector of numbers of points falling within leaf nodes \( LN \), vector of numbers of starting positions of points in leaf nodes \( PN \), number of leaf nodes \( n_l \) and number of points falling within the leaf nodes \( n_s \) (at all levels)

Initialization: Set \( n_p \) (representing number of identified points in leaf nodes) to 0 and set \( n_q \) (representing number of identified leaf nodes) to 0.

For \( k \) from 1 to \( M \):
1.1. Transform point dataset \( P \) to key set \( PK \) using Z-ordering at level \( k \)
1.2. Sort by key using \( PK \) as the key and \( P \) as the value
1.3. Reduce by key using \( PK \) as the key and copy the unique keys to \( UK \) and numbers of the same key in each key group into \( UN \)
1.4. Classify each quadrant (corresponds to a key in \( UK \)) based on whether the numbers in \( UN \) is above (set to 0) or below (set to 1) the threshold \( K \) and copy the result to a boolean vector \( SIGN \) by using Transform
1.5. Identify points that are within or not within the quadrants to be pruned based on \( UN \) and \( SIGN \) by using Expand and output the result to a boolean vector \( INDICATOR \).
1.6. Padding level \( k \) keys in \( UK \) to the finest level using in-place Transform (with left-shifting).
1.7. Copy the identified leaf node keys to \( LK \) and number of points in the leaf nodes to \( LN \), starting at \( n_q \), by using Copy if based on \( UK \), \( UN \) and \( K \) also set \( n_l \) to the number of identified leaf nodes at the level
1.8. Exclusive scan \( LN \) by taking \( n_p \) as the initial value and output the result to \( PN \).
1.9. Copy all points in \( P \) that are in the identified leaf nodes to \( PL \) and those that are not in the identified leaf nodes to \( PQ \) using Copy if based on \( INDICATOR \).
1.10. Combine \( PL \) and \( PQ \) using Copy by placing \( PL \) ahead of \( PQ \)
1.11. Compute the number of points \( (n_s) \) that fall within identified leaf nodes by using Reduce based on \( LN \).
1.12. Increase \( n_p \) by \( n_s \) and increase \( n_q \) by \( n_l \).
The first part is designed to generate leaf quadrants in a BFS order. The algorithm is presented in Fig. 3. We also refer to Appendix 2 for a running example for this part (note that it is the same example that we have used to introduce the CSPT-P tree in Fig. 1 and Fig. 2). In Fig. 3, the names of parallel primitives are bolded and underlined and the variables (either a vector or a scalar) names are bolded and italicized for easy interpretation. Steps 1-3 in Fig. 3 are used to sort points (stored in P) based on their level k Morton codes (used as keys, stored in PK) and count the numbers of points (stored in UN) associated with the unique keys (stored in UK). The points are also sorted based on the keys so that they can be reordered later (Steps 1.9 and 1.10) and get ready for the next level. Steps 1.4 and 1.5 are used to identify leaf quadrants and the points associated with the quadrants. Note that the SIGN vector indicates whether a quadrant is a leaf one and the INDICATOR vector indicates whether a point belongs to a leaf quadrant. For each number in UN (assuming $n$), which records the number of points with the same level-k key, the boolean value in the SIGN vector at the same position will be replicated $n$ times in INDICATOR. This is done by using the Expand parallel primitive that has been implemented by combining a Scatter and a Gather primitive (to be detailed next).

To better illustrate how the parallel primitives work together to identify tree leaf nodes, let us consider the following SQL statement “SELECT * FROM T WHERE #key IN (SELECT #key FROM T GROUP BY #key HAVING COUNT (#key)) > #K”. The statement selects individual tuples that satisfy a count-based group condition and does what we want in Step 1.5. While it is straightforward to output individual tuples whose #key values are in the resulting single-attribute relation of the sub-query with the group by/having clauses on CPUs, it is neither convenient nor efficient to perform set membership tests on GPUs. Actually we do not have to due to the relationships among UK, UN and PK. Obviously UK and UN are group-related (when referencing to the SQL statement). The evaluation results of the having condition should be a boolean vector (i.e., SIGN that has the same length as UK and UN. Since UK and PK has the same key order, when mapping UK back to PK, each boolean value at SIGN[i] will repeat exactly UN[i] times and the vector of such boolean values (INDICATOR) exactly indicates whether a key in vector PK satisfy the group-based criteria (i.e., the condition specified in the having clause in the example SQL statement). Now the problem translates into how to generate the INDICATOR boolean vector from the SIGN boolean vector and the UN integer vector. This actually can be done using four parallel primitives that are introduced in Appendix 1. First, an exclusive scan is performed on UN to compute the group boundaries. Second, a Scatter primitive is used to scatter the group boundary values to proper positions in a temporal vector (say VT) that has the same size as PK/INDICATOR. Third, a Scan primitive using the max associative function is performed to propagate the boundary values in VT to positions within group boundaries. Finally, a Gather primitive is applied to update the values in INDICATOR by the values in SIGN using VT as the map, i.e., the $i^{th}$ element in INDICATOR is assigned to the value of SIGN[VT[i]].

Step 1.7 copies the identified leaf keys and the corresponding numbers of points to two new vectors (LK and LN) which will be used in the next part to eventually construct a full quadtree. The purpose of Step 1.6 is to pad level k keys and make them a level M key by bit-wise left shifting so that all leaf keys will have the same length at the beginning of the first step in the second part. Step 1.8 calculates the numbers of points that are associated with all the level k leaf nodes and the expected first point positions on the rearranged point array. Steps 1.9 and 1.10 actually rearrange the points by moving identified leaf points to the left and the rest of the points to the right so that the next round only needs to process the unprocessed points. Finally Step 1.11 computes the total number of leaf points identified at the level k and Step 1.12 records the level boundaries by adding the numbers of identified leaf nodes and the numbers of identified leaf points.

The second part of the CSPT-P tree construction takes the following output vectors of part1 as the inputs: leaf node keys (LK), numbers of points associated with leaf node keys (LN), positions of the first points in the quadrants corresponding to leaf nodes (PN) and the rearranged point vector (P). In addition, the scalar values to delimit the level boundaries in the vectors are also needed. We note that no further modifications on the PN and P vectors are needed in the second part and they are accessed (read-only) in the second part to construct the final CSPT-P tree. Similar to part 1, we have listed the steps in Fig. 4 and an illustrative example is provided in Appendix 3. We note that only the first two level processing is illustrated Appendix 3 due to space limit. Step 2.1 takes leaf nodes identified in the first part (i.e., the LN vector) that are relevant to level k nodes, including both leaf nodes and non-leaf nodes, to compute the numbers of the immediate children under each of the quadrants. This can be done by transforming LK to NK after truncating LK keys (Morton codes) at the level k+1 and then count the unique numbers of level-k keys. Since the elements to the left of the level k-1 boundary in the LN vector are leaf nodes above level k and will not contribute to the level k nodes (recall that the leaf nodes follow a BFS order), we can safely exclude them from the processing at the level k. It might be attempting to use only level k+1 leaf nodes to construct level k nodes to further save computation, however, this would give wrong results due to the BFS ordering. For example, assuming that we have only two leaf nodes represented as X.1 and X.2.3 where X can be any valid space partition tree path (prefix) and we want to build a subtree rooted at a node corresponding to X (assuming level l). If we only look at the leaf node at level $l+1$, since X.1 is the only leaf node at the level, our construction algorithm will conclude that the subtree root has only one child node. However, it is easy to see that X.2 is also a child node of X due to the existence of X.2.3 which is sorted at level $l+2$. As such, we need to examine all leaf nodes from level k to the last level in LN (M).

The resulting NK vector from Step 2.1 may have duplicates which need to be removed before counting. We first sort NK (Step 2.2) to let the same NK keys appear consecutively and then apply the Reduce by_key primitive to get a unique key vector MK (Step 2.3). Computing the numbers of the immediate child nodes under each level k node is achieved using the following three steps. First, in Step 2.4, all level k+1 keys are truncated to level k keys through right-shifting and the results are recorded in TK. Second, in Step 2.5, all the level k leaf nodes in LK are appended to TK. Third, similar to Step 2.2, keys in TK are sorted in Step 2.6. Finally Step 2.7 counts the numbers of the immediate child nodes for all level k nodes by applying Reduce by_key primitive again. The level k keys are recorded in WK and the numbers of immediate child nodes are recorded in NW.
The remaining steps in the second part are related to computing the positions of the first child nodes for non-leaf quadtree nodes and filling the data fields in the leaf nodes derived from the first part. The first task can easily be accomplished by using a parallel prefix sum on the vector of numbers of immediate children of level k nodes (Step 2.8) where the results are stored in vector NX. Of course we need to add the number of quadtree nodes that have already been derived at level k to a tally and use the tally as the initial value of the prefix sum. The second task is more involved as the leaf nodes identified in the first part are only a subset of the quadtree leaf nodes at the corresponding levels (say level k). In order to “replace” the leaf nodes generated in the second part with the proper information of the leaf nodes generated in the first part, we have to establish the correspondence. While this may be an easy task in a serial implementation as we only need to process the two sources sequentially, it becomes nontrivial in parallel implementation as each processing unit (thread) needs to know the exact position to retrieve data items from the two sources simultaneously. Fortunately, the scatter primitive serves the right purpose. To scatter the numbers of points and the starting positions in the point vector for all leaf nodes at level k identified in the first part, we first copy leaf node positions (the corresponding elements in NX are 1s) to NZ and use NZ as a map for scattering. For the i th element in LN (or PN), if NZ[i]=j then LN (or PN) will be written to the jth position of a resulting vector LM (or PM). Note that LM and PM have the same lengths as WK, NW and NX. These five vectors can then be processed in parallel to construct CSPT-P tree nodes by using a Transform primitive. Note that the LM and PM vectors will have NO_DATA value if their positions correspond to non-leaf nodes. On the other hand, the number of child nodes for a leaf node in NW is always 1 and the first child node positions for a leaf node is meaningless. This characteristic makes it possible to reuse the number of child nodes and the first child position fields of a CSPT-P tree leaf node to carry valuable information of the points that are associated with it.

<table>
<thead>
<tr>
<th>Part 2: Build CSPT-P Tree level by level</th>
<th>Inputs: the outputs of the first part</th>
<th>Outputs: vector of quadtree nodes Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>For k from 1..M-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 Transform LK into NK by truncating keys in LK from ( \sum n_l[k+1] ) to ( \sum n_l[M] ).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2 Sort NK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3 Reduce by key using NK as the key and copy the unique keys to MK.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4 Generate level k keys by shifting all key values in MK to the immediate upper level and output them to TK using Transform.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5 Append the leaf node keys of level k in LK (from ( \sum n_l[s-1] ) to ( \sum n_l[s] ) where s from 1 to k) to TK by using Copy.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6 Sort TK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.7 Reduce by key using TK as the key and copy the resulting unique keys to WK and the numbers of the same key in each key group to NW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.8 Exclusive scan on NW and output the result to NX to compute the first child position of a quadtree node represented by a key in WK.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.9 Output the positions of the elements in NW into NZ for those whose values are 1 (leaf nodes) using Copy_if.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.9 Scatter LN and PN to LM and PM, respectively, using NZ as the map.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.10 Create the quadtree node vector Q from WK, NW, NX, LM and PM at level k by using Transform.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finalization: append the last level leaf nodes to Q from LK, LN and PN by using Transform.

5 EXPERIMENTS AND RESULTS

To evaluate the feasibility and efficiency of our parallel CSPT-P tree construction algorithm, we have tested our implementation on the pickup locations of roughly 170 million taxi trip records in 2009 (after data cleaning to eliminate records that do not have valid latitude/longitude coordinates). We have used 32-bits Morton codes and set block capacity K to 2000 in this study. The fan-out of our CSPT-P tree is set to \( 2^8 = 16 \) and there are at most 8 levels in the resulting tree. All experiments are performed on a Dell T5400 machine equipped with an Nvidia Quadro 6000 GPU device. Although the dual quad-core CPU (Intel Xeon E5405) has a total of 8 cores running at 2.0GHZ clock rate, only one core is used for comparison purpose. The GPU device has 448 GPU cores running at 574 MHZ. The device also has 6 GB DDR5 memory with a bandwidth of 144 GB/s. Our GPGPU implementation is compiled with CUDA 4.0 (with computing compatibility set to 2.0) and uses Thrust 1.6.0. For comparison purpose, we have also provided a straightforward serial CPU implementation. The CPU code is compiled using g++ 4.4.3 under Ubuntu 10.04 with –O2 optimization for speed to ensure fair comparison.

Our results have shown that the runtimes of the primitives in the first part dominate. As such, we will show the runtimes of each level for groups of primitives used in the first part but combines the runtimes of primitives in the second part. To test the scalability, we have run the GPGPU implementation multiple times by experimenting on data for 1, 2, 3, 4, 6, 8 and 12 months in 2009. The results are tabulated in Table 1 (all times are measured in milliseconds). From the table we can see that the tree building times that include the runtimes of all steps in the second part are fairly small. Looking into the output logs corresponding to the using 12 month data test case, we found that our implementation generates 483,529 quadtree nodes (including 30,245 non-leaf nodes) in just 31.76 milliseconds and has achieved 15 million nodes per second rate. This indicates that our implementation can be used to generate tree indices with much larger numbers of nodes. From the results we can also see that both the total times and the sorting times increase linearly with the numbers of points which has clearly demonstrated the scalability of the design and the implementation.

Our baseline CPU implementation for comparison has three parts to produce the same data fields and ordering as in our GPGPU implementation. The first part is to build a CSPT-P tree itself. The second part is to traversal the tree in a Depth-First Search (DFS) manner to gather numbers of points that fall under each tree node so that quadrants which less than K points can be identified when traversing the tree again in a BFS order. When feeding the 12 months’ taxi trip pickup locations to the baseline implementation, it takes 162,004 milliseconds to complete the tree construction, 3,166 milliseconds to traverse
the tree in DFS order to gather number of points for all tree nodes and 46.50 milliseconds to traverse the tree in BFS order to identify and output quadrants that have fewer than $K=2000$ points. Since once the tree is constructed after the first part is done, the indexing tree can already be used for spatial query processing without part two or three, to ensure fair comparison, we will only use the runtime of the first part to compare with the GPU implementation. We can see that the runtime of the first part of the CPU serial implementation is about 23 times of the total time of the GPU implementation. In other words, we have achieved about 23X speedup (162004/7016). Note that the total time in our GPU implementation has already included point data transfer time to GPU for fair comparison. Furthermore, while the GPU code is a little complex than the CPU code, the numbers of code lines between the two are comparable. In fact, after removing comments, debugging outputs and timing measurements, each of the steps listed in Fig. 3 and Fig. 4 corresponds to a single line, although additional lines of code to implement simple functors (to be used with primitives such as Transform and Copy_if) are needed. In general, we believe that our primitives based GPU implementation achieve a good tradeoff between efficiency of utilizing parallel hardware and coding complexity which is attractive to a variety of domain applications.

<table>
<thead>
<tr>
<th>Months</th>
<th>1-1</th>
<th>1-2</th>
<th>1-3</th>
<th>1-4</th>
<th>1-6</th>
<th>1-8</th>
<th>1-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>#of points</td>
<td>13,887,620</td>
<td>27,079,723</td>
<td>41,784,081</td>
<td>55,383,596</td>
<td>84,035,490</td>
<td>111,127,610</td>
<td>168,898,952</td>
</tr>
<tr>
<td>Total Time</td>
<td>546.07</td>
<td>1033.93</td>
<td>1634.55</td>
<td>2215.26</td>
<td>3394.73</td>
<td>4605.97</td>
<td>7016.45</td>
</tr>
<tr>
<td>Data transfer time to GPU</td>
<td>56.91</td>
<td>109.80</td>
<td>166.47</td>
<td>223.11</td>
<td>337.77</td>
<td>446.59</td>
<td>678.64</td>
</tr>
<tr>
<td>Key Transformation time (Step 1.1)</td>
<td>24.91</td>
<td>50.57</td>
<td>79.96</td>
<td>108.36</td>
<td>167.42</td>
<td>224.41</td>
<td>347.45</td>
</tr>
<tr>
<td>Key Sorting time (Step 1.2)</td>
<td>264.88</td>
<td>542.04</td>
<td>855.03</td>
<td>1170.47</td>
<td>1806.91</td>
<td>2496.52</td>
<td>3795.23</td>
</tr>
<tr>
<td>Key Reducing Time (Step 1.3)</td>
<td>39.78</td>
<td>77.20</td>
<td>119.18</td>
<td>161.11</td>
<td>246.35</td>
<td>329.22</td>
<td>504.93</td>
</tr>
<tr>
<td>Leaf node/point Identifying Time (Step 1.4-1.8)</td>
<td>56.52</td>
<td>107.25</td>
<td>162.07</td>
<td>218.37</td>
<td>335.67</td>
<td>447.05</td>
<td>683.80</td>
</tr>
<tr>
<td>Point shuffle Time (Step 1.9-1.10)</td>
<td>73.86</td>
<td>145.66</td>
<td>227.12</td>
<td>306.94</td>
<td>471.63</td>
<td>633.31</td>
<td>974.64</td>
</tr>
<tr>
<td>Tree building time (all steps in the second part)</td>
<td>29.21</td>
<td>21.41</td>
<td>24.72</td>
<td>26.90</td>
<td>28.98</td>
<td>29.87</td>
<td>31.76</td>
</tr>
</tbody>
</table>

### 6 CONCLUSION AND FUTURE WORK

In this study, we have presented a primitives-based design and implementation on indexing large-scale point data on GPGPUs. More specifically, we have designed a Constrained Space Partitioning Tree for Point data (i.e., CSPT-P tree) that is suitable for parallel construction and implemented the method construction algorithm on GPGPUs on top of the Thrust parallel library. Experiment results have demonstrated a 23X speedup on a large-scale point dataset that consists of about 170 million taxi pickup locations in NYC in 2009, when compared with a baseline serial CPU implementation. The design and implementation represent a good tradeoff between the efficiency of utilizing parallel hardware and coding complexity and we believe the feature is attractive to a variety of domain applications beyond geospatial data management.

There are quite a few directions that we want to pursue for our future works. First, we would like to compare the performance of the primitives based implementation with a native CUDA implementation that can potentially utilize GPUs more effectively, such as increasing degree of parallelization and better utilization of GPU memory, to understand the performance gaps between the two. Second, we would like to run our implementations on other parallel hardware architectures that are supported by the Thrust library such as multicore machines and further evaluate the tradeoffs among various factors including efficiency, complexity and portability. Finally, we plan to extend the framework that we have used in this study to other types of spatial indexing data structures.

### REFERENCES

11. OGC SFS. http://www.opengeospatial.org/standards/sfs
Appendix 1: Parallel Primitives

Although naming conventions might differ slightly under different contexts and software implementations, since our implementation is based on the Thrust library [10], we next introduce the primitives that we have used in our design using the Thrust terminology.

(1) Reduce and Reduce by key. Reduce is used to simplify a vector/array to a scalar value. For example, reduce([-3,2,4]) → 11. While the summation is frequently used in reductions, Thrust allows using a user defined associative binary function for tailored summation, such as determining the maximum entry or computing bounding boxes of points. Reduce by key is a generalization of Reduce to key-value pairs based on groups where consecutive keys in the groups are the same. For example, reduce([1,3,2][2,1,3,4]) → ([1,3,2][2,4,6]). In this research, Reduce by key has been extensively used to compute numbers of points or tree nodes that have the same keys.

(2) Scan and Scan by Key. The Scan primitive computes the cumulative sum of a vector/array. The Scan primitive can also take a user defined associative binary function. Both the inclusive and exclusive scans are available. For example, exclusive_scan([-3,2,4]) → ([0,3,5]) while inclusive_scan([-3,2,4]) → ([3,5,9]). Similarly, Scan by Key works on consecutive key groups instead of a whole vector/array. In this research, Scan by Key is extensively used to compute the positions of entries in a vector after applying Reduce by key which outputs numbers of entries with same keys.

(3) Copy and Copy_if. The functionality of the two primitives is self-evident. In this research, we use Copy to move groups of entries from one location to another, mostly within a same vector. The Copy_if primitive is mostly used for identifying points and keys (tree nodes) that satisfy certain criteria and output the identified entries to a new vector for further processing.

(4) Gather and Scatter. Gather copies elements from a source array into a destination range according to a map and Scatter copies elements from a source range into an output array according to a map. For example, Gather([-3,0,2],[4,7,8,12,15]) → ([12,4,8]) and Scatter([-3,0,2],[12,4,8],[*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.]

Appendix 2: A running example of part one of CSPT-P tree construction

For the example shown in Fig. Appendix-2 (best viewed in color), at the level-1, there are three quadrants that have points. The lower-left quadrant is identified as a leaf node as the number of points that fall within the quadrant is 11 which is lower than the predefined threshold (assuming K=20 in this example). After the first part completes, 1 leaf node (2) and 11 leaf points are identified at the first level and 2 leaf nodes (4 and 7) and 18 leaf points (9+9) are identified at the level 2. All the rest of the leaf nodes (6 in total) are identified at level 3. The prefix sum results shown at the bottom of the figure represent the starting positions of the leaf points assigned to the 9 leaf nodes on the rearranged point vector.

Appendix 3: A running example of part two of CSPT-P tree construction

For the example shown in Fig. Appendix-3 (best viewed in color), since there are three quadrants that are non-empty at the first level, the root quadtree node has 3 nodes and the first child node position is 1 (always). At the next level (k=2), the two quadrants at the top have 3 and 2 child nodes, respectively. Note that the quadrant corresponds to node 2 is not processed at level 2 as it has already been processed at level 1 and should be excluded.

At the level 2, as there are 4 tree nodes for levels 1 and 2, we use it as the initial value for inclusive prefix sum on the NW vector of (3, 2). The derived first child node positions for the two quadtree nodes at level 2 are thus 4 and 7, respectively. These data are used to fill the data files of the three level-2 tree nodes (ref to Fig. 1 for the data fields of the rest tree nodes).
Fig. Appendix-2: Illustrations of steps in part one (detailed in Fig. 3) using an example

Fig. Appendix 3: Illustrations of part two steps (detailed in Fig. 4) using an example