Efficient Quadtree Construction for Indexing Large-Scale Point Data on GPUs: Bottom-Up vs. Top-Down

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

Point location data are rapidly growing in volume, velocity and variety. GPU-acceleration is attractive for managing large-scale point data. In this study, we introduce a new bottom-up approach to constructing quadtrees for indexing large-scale point data on GPUs, which is significantly different from previous works that typically adopt a top-down strategy. In addition to sorting points to be indexed only once based on their Morton codes, our proposed bottom-up approach adopts a data parallel design and parallel primitive based implementation, which makes a sensible tradeoff between efficiency and complexity in principle and often results in more efficient implementations in practice.

We also extend our previous work [1] on identifying leaf quadrants from large-scale point datasets by repetitively partitioning the indexing space into quadrants until the number of points in each quadrant is less than a threshold. Although our previous work was designed for parallelizing spatial joins on GPUs, the extension is able to construct the quadtree index end-to-end in a top-down manner. As this top-down approach is more efficient than the quadtree construction code provided as a sample in Nvidia CUDA SDK, we use it as a strong baseline to compare with our newly proposed bottom-up approach and explore some aspects of duality between the top-down and the bottom-up approaches.

Experiments show that the bottom-up approach is capable of indexing approximately 170 million taxi pickup locations in New York City (NYC) in less than 200 milliseconds and is 3.4X and 4.9X times faster than the top-down approach with and without including CPU/GPU data transfer time, respectively. The work reported in this paper is part of an effort to integrate spatial data management functionality into a GPU-accelerated data management system to support both relational and spatial data in an integrated manner.

1. INTRODUCTION

Comparing to polygon and polyline data in geospatial applications, the volumes of point data have been increasing very fast due to significant progresses of sensing techniques and advanced processing tools in recent years. For example, in contrast to point locations captured by consumer GPS devices whose temporal resolution is in the order of a few to tens of seconds per record, locations derived from high-resolution cameras using Deep Learning (DL) inferencing tools can have a frame rate in the order of a few tens of frames per second and there can be tens to hundreds of detected objects in a single image. The combined point location data volume increases for moving vehicles can be 2-3 orders higher. The same tends also apply to human poses under different application scenarios. While parallel hardware and distributed systems are emerging to handle such "Spatial BigData", one of the key challenges remains is to understand the inherent characteristics of data access patterns in indexing large-scale point data for various types of spatial queries, such as range queries and spatial joins, on different parallel hardware and distributed computing platforms.

As Graphics Processing Units (GPUs) hardware are getting into mainstream and General Purpose Graphics Processing Unit (GPGPU) technologies become mature, there are increasing research and application interests on in-memory data managements on GPUs, such as the open source cuDF project from Nvidia [2], CoGaDB [3] and OmniDB [4]. However, none of these GPU databases has the capability to support spatial data management yet. On the other hand, while a few research projects have developed techniques and prototype systems for managing large-scale spatial data on GPUs and reported impressive performance [5] [6], they are yet to be widely applied due to the shear amount of software engineering work to integrate these techniques into end-to-end systems with desirable level of usability, in addition to performance.

In this study, built on top of our previous work on managing large-scale point data on GPUs [1], we propose to revisit the interesting topic from both a parallel design perspective and its implementations on modern GPUs while keep integrating with mainstream GPU data management systems such as cuDF in mind. In addition to extending our previous technique [1] to construct quadtrees end-to-end for point indexing, which we call a top-down approach, to form a strong baseline for comparison, we have proposed a new bottom-up approach that is more efficient not only in terms of end-to-end runtimes but also with respect to memory footprint. We analyze the two quadtree construction approaches and highlight some key findings with respect to data accesses that largely dominate the performance differences between the two approaches. While our techniques currently adopt a non-overlapping space partitioning scheme for quadtree-alike indexing, we believe many of the discussions can be applied to data-driven spatial indexing, such as R-Trees [7]., as well.

The rest of the paper is arranged as follows. Section 2 is the background, motivation and related work. Section 3 introduces the baseline approach by extending the work we have reported in [1]. Section 4 presents the new bottom-up approach for indexing large-scale point data. Section 5 provides experiments and discusses results using taxi pickup locations in NYC. Finally, Section 6 is the conclusion and future work directions.

2. BACKGROUND, MOTIVATION AND RELATED WORK

Spatial indexing plays a key role in spatial data management [7]. For a 2D range query, aka. Window Query, by exploiting spatial indexing, the complexity can be reduced from O(*n*) to O(log*n*) or even O(1). Probably more significantly, for spatial joins that involve two input datasets, the complexity can be reduced from O(*m***n*) to O(*m**log*n*)/O(*n**log*m*) or even lower, where *m* and *n* are the numbers of records in the two input datasets in a spatial join.

Indexing on polyline or polygon data typically approximates complex polylines and polygons using Minimum Bounding Boxes (MBBs) and subsequently indexes the MBBs. While technically feasible, it would be too costly to degenerate points as MBBs and then apply techniques such as R-Trees [7] for indexing for the purpose. Although various hashing techniques have been developed for generic multidimensional point data (see e.g. [8] [9] for GPU-based construction techniques), hierarchical indexing structures, such as space partition based kd-trees and quadtrees, remain popular for indexing geospatial point data that typically have two or three dimensions, partially for interpretability and visualization. In this study, we focus on multi-level space partition based point indexing for 2D geospatial point locations on GPUs and we refer to the previous works for spatial indexing on polylines and polygons on GPUs [10] [11] [12].

While numerous spatial indexing methods have been proposed in the past few decades [7], most of them are designed for CPUs with a single processor based on a serial computing model. The past few years have witnessed rapid advances of parallel hardware, such as multi-core CPUs without and with SIMD (Single Instruction Multi Data) extensions [13] and GPUs with thousands of cores and increasingly larger memory capacities [14]. While multi-core CPUs allow coarse-grained parallelisms and can be relatively easily applied to parallelizing serial spatial indexing techniques, it is quite technically challenging to fully exploit SIMD instructions that are supported by Vector Processing Units (VPUs) inside CPUs due to their stringent parallel programming models [13]. In contrast, the emergence of CUDA technologies supported by Nvidia GPUs have provided a flexible parallel computing platform, a set of easy-to-use APIs, and a user-friendly and portable parallel library (i.e., Thrust [15]) that consists of a set of highly efficient parallel primitives to exploit the massive data parallel computing power on GPUs [14].

There has been a steady growth of interests in using GPUs for spatial data management in the past few years (we refer to papers in a 2014 special issue of ACM SIGSPATIAL Special [10] [6] for more comprehensive reviews), in parallel with using GPUs for relational data management [2] [3] [4]. In a way similar to the fusion of Relational Database Management System (RDBMS) and Geographical Information System (GIS) in late 2000s after major database vendors supported spatial data, we believe there is a technical trend in developing GPU-accelerated data management systems that support both relational and spatial data. Among all spatial data types, point data might be the first one that is fully supported in such systems due to practical popularity, data layout regularity (fixed length for 2D/3D points) and technical maturity, in a way similar to that, point data seemed to be first supported by NoSQL databases such as Cassandra before more complex spatial datatypes were supported [16].

It is obvious that performance is the main driving force for GPU accelerated data management on large-scale data. While indexing is optional for moderate size datasets by fully exploiting GPU hardware capabilities, indexing becomes indispensable for large-scale data. Spatial indexing on point data is challenging largely due to high data volumes and significant irregular data accesses and data movements required for indexing. GPUs are traditionally featured with large number of processing units, high memory bandwidth and user managed shared memory/caches, which are desirable from a data management perspective. However, high memory latency and limited cache capacity have imposed significant technical challenges, in addition to complexity and programmability, for such purposes [14]. In our previous studies, we have advocated for parallel primitive based designs and implementations for spatial indexing and spatial queries (e.g. [1] [17] [18] [19]). Parallel primitives supported by GPU hardware, e.g., the Thrust library through CUDA SDK [15] [14], provide a desirable isolation between our spatial data management applications and GPU hardware details and facilitate a well-justified tradeoff between portability and efficiency. The set of 7 parallel primitives (with variations) used in this study are listed in the Appendix for a quick reference.

Among various parallel primitives we have exploited, sorting plays a key role in spatial indexing to move spatially adjacent points close to each other and generate indices for the underlying data. Fortunately, sorting is among the most well-studied parallel primitives on GPUs and its implementations have been continuously improving and fine-tuned [20] [21]. Modern GPUs are capable of sorting hundreds of millions or even billions of data items per second which largely contributes to the overall high performance of GPU-based spatial indexing techniques. Nonetheless, as detailed in Section 3 and Section 4, utilizing the parallel primitives (including sorting) in different ways may significantly impact the performance of spatial indexing applications (and likely many other applications), which motivates many of the discussions in this work. As detailed in Section 5, the new bottom-up technique, while exploiting the same set of parallel primitives, has achieved nearly 5X speedup on indexing approximately 170 million points when compared with the top-down approach.

Kd-trees and quadtrees are quite similar on indexing point data, as both of them belong to space partition based spatial indexing techniques. While quadtrees have been popular in the spatial databases and GIS communities, their realizations on GPUs appeared much later. Kd-trees have been popular in the computer graphics community for applications such as ray tracing, which can be dated back to the pre-GPGPU era [22]. To the best of our knowledge, the work reported in [23] is among the earliest on kd-tree construction on GPUs. On the other hand, constructing kd-trees on point data for distance-based pruning of search space completely on GPUs did not appear until recently [24]. Nevertheless, many design strategies and implementation techniques can be applied to both kd-tree and quadtree constructions on GPUs.

The works in [25] [26] [27] [28] follow the idea of treating quadtree construction on GPUs to a level-by-level bucket sort problem where quadtree nodes are considered as buckets and points to be indexed are sorted into the buckets based on the quadrants that the points fall within. In particular, [25] adopts a CPU-GPU hybrid approach where each thread processes a quadtree node and loops over the points under the node for subsequent subdivisions. The first few levels of quadtree construction are performed on CPUs as the number of quadtree nodes is small and the degree of parallelism is low. The technique switches to GPUs only when a sufficient number of quadtree nodes has been created and GPU parallel computing capacity can be effectively utilized. However, as each CPU thread needs to process a large number of points for the upper level quadtree nodes, the within-node parallelism is not effectively utilized and the CPU processing time could dominate. For the rest of the quadtree nodes constructed on GPUs, it is likely that the underlying bucket sort is performed by a single GPU thread which can be inefficient. The two major sources of design and implementation drawbacks make the technique rather inefficient and the authors reported a runtime of 20 seconds for indexing 15 million points on an unspecified Nvidia GPU around 2011.

The work reported in [26] improves [25] by exploiting block-level parallelism and letting a block of GPU threads process all the points indexed by a quadtree node for subdivisions. The technique can significantly increase the degree of parallelism for constructing quadtree nodes, including top-level ones. This makes an all-GPU based solution possible. However, there are two new sources of overheads. The first is the cost of coordination among threads in a block when sorting points. The second is that threads in warps may be underutilized when the number of points within a block is less than GPU warp size (typically 32). The authors proposed to switch to single-thread serial construction in this case instead (as in [25]). According to the performance report in [26] , the technique is able to index 80 million points in about 50 seconds on an Nvidia FX 3800 GPU which seems to be more efficient than the results reported in [25].

CUDA SDK provides a sample code on quadtree indexing for points starting from version 5.0 for GPUs that supports Compute Capability 3.5 as a demonstration of dynamic parallelism [27]. The design behind the code is similar to [26] in serval aspects but with better block/warp level reduction and scan support for bucket sort. With hardware support of dynamic parallelism, the end-to-end process can now be performed using a single recursive kernel function invocation. New kernels are launched dynamically by threads that are processing quadtree nodes that need subdivisions. Four warps in a block are used to bucket-sort the points where each warp is assigned to process a quadrant. Warp level hardware intrinsic functions, such as voting and shifting [29], are aggressively exploited for efficiency. However, the policy on processing a quadrant using a single warp may also limit the degree of parallelization and subsequently affect efficiency, especially during the construction of top-level nodes where a large number of points need to be processed. We also note that the publically available code uses two identical arrays to store points before bucket sort and after bucket sort, respectively. The two arrays are swapped when processing two consecutive levels during quadtree construction, which makes the implementation less attractive comparing with those that support in-place sort. Different from [26] that only expands non-empty quadtree nodes level-by-level, the CUDA SDK sample code pre-allocates memory for a full quadtree (i.e., a pyramid) of depth k with a memory footprint of $(4^{(k+1)}-1)/3$ before the construction process begins, which is memory inefficient especially when depth *k* is large. A more recent work [28] overcomes this memory inefficiency by allocating memory only for the exact number of non-empty quadtree nodes. This is achieved by first counting the number of such non-empty quadtree nodes using hardware units that support atomic operations (e.g., atomicAdd [29]) before actually outputting data. However, the number of hardware units that support atomic operations is limited on commodity GPUs and the performance is likely to be bottlenecked by the hardware units when indexing large numbers of points.

The implementations of all of the four techniques discussed above seem to be based on the plain CUDA programming syntax directly, which is expected to maximize performance but may not be easily achievable due to implementation complexity. They share the similar idea of using bucket-sort for quadtree construction and they also suffer from insufficient degree of exploitable parallelization due to the nature of the bucket-sort design. First of all, regardless of whether parallelizing quadtree node construction is at the thread level, warp level or block level, a thread needs to loop over potentially a large number of points. We note that the distribution of the point counts can be skewed among quadrants. Real world point data are typically unevenly distributed, such as taxi pickup locations in NYC where most of the locations are in downtown and middle town areas, especially around hotspots in these areas. The skewed distribution makes parallelization very difficult due to unbalanced workloads for parallel processing units. Second, whether utilizing dynamic parallelization or not, as a quadtree typically has only four child nodes at most, exploiting node level parallelism generally results in insufficient workload when constructing the first few levels of a quadtree. While overlapping multiple streams of GPU jobs may potentially increase the overall GPU utilization, it does not help end-to-end runtime when constructing a single quadtree. Third, while bucket-sort is conceptually simple, its GPU implementation is not as extensively fine-tuned as radix sort on GPUs [21]. Implementing bucket sort at the block level using a double array approach, which seems to be adopted in the previous works, results in excessive non-coalesced GPU global memory copy operations which could be costly with respect to both memory footprint and runtime.

Our preliminary work on identifying quadrants with numbers of points less than a predefined threshold to balance workloads in point-in-polygon test based spatial joins on GPUs [1] predates or in parallel with the works discussed above [25] [26] [27] [28]. Even though the technique only identifies such quadrants level-by-level without actually creating a quadtree, they share quite some commons in design. However, different from these works, our work [1] exploits both node and point level parallelisms and adopted a parallel primitive based strategy to make a sensible tradeoff between efficiency and simplicity. In this study, we extend our previous work to complete a GPU-accelerated quadtree construction technique and refer it as our top-down approach for point indexing (Section 3). As the additional phase that constructs the quadtree from leaf quadrants takes very little additional runtime, the efficiency of the technique is largely determined by the first phase on identifying leaf quadrants. While we defer the presentation of the technique to Section 3, the parallel primitive based implementation avoids several key issues encountered by other works on quadtree-based indexing for point data directly using plain CUDA syntax. When comparing to the implementation of the CUDA SDK sample code [27] (the source code of the other three works is not publically available and it is non-trivial to re-implement), the top-down approach can achieve 2.5X higher performance (Section5), likely due to better exploitation of parallelisms in sorting points to appropriate quadrants and efficient implementations for parallel primitives, in a way similar to what we have reported in [19] on constructing multi-attributed quadtrees for complex polygons using parallel primitives. Furthermore, by relaxing certain requirements imposed by the top-down approach, we have developed a new bottom-up approach (Section 4) and we use the top-down approach as a baseline to compare with the newly proposed bottom-up approach. We next present the top-down and the bottom-up approaches in Section 3 and Section 4, respectively.

3. GPU QUADTREE DATA LAYOUT AND THE TOP-DOWN APPROACH

Before presenting the top-down approach, we first introduce the data layout of the quadtree structure that was designed for the top-down approach but can also be applied to the bottom-up approach. For notation convenience, when introducing the proposed Structure-of-Array (SoA) quadtree data layout, array names are bolded and italicized when they are first introduced and they will only be italicized when they are subsequently referred to. When introducing the algorithms using parallel primitives, parallel primitives are underscored and their input/out variables are both bolded and italicized while temporal variables are italicized only in figures. All of the inputs/output variables and parallel primitives are only italicized in their accompanying text for clarity. Some words in text are occasionally bolded (but not italicized) for emphasis purposes. The conventions are applied to Section 4 as well.

3.1 SoA-based Quadtree Data Layout

The structure is similar to that in the previous works [25] [26] [27] [28] in the sense that quadtree nodes are laid out as an array in a level-by-level order, i.e., Breadth First Search or BFS. However, our quadtree structure adopts a Structure of Arrays (SoA) rather than Array of Structures (AoS) strategy (as in the CUDA sample code) for better coalesced memory accesses on GPUs. The AoS structure in our quadtree consists of four arrays with the same length. The Morton (Z-Order) code [7] array, termed as the ***key*** array, stores the Morton codes of quadrants. The boolean leaf indicator array, termed as the ***indicator*** array, stores 1/0 values to indicate whether the respective quadtree node is a leaf node or not. The first child/point position array, termed as ***f_pos*** array, stores the positions of the first child node in the quadtree SoA for a non-leaf node **or** the positions of the first points that fall within the quadrants that are indexed by a leaf quadtree node, depending on the corresponding elements in the *indicator* array. The last one, termed as ***length*** array, stores the numbers of child nodes of non-leaf nodes **or** the numbers of points of leaf nodes, again depending on the *indicator* array elements. Note that the *length* array and the *f_pos* array hold the information related to both quadtree nodes and points. The combination is possible due to the fact that leaf-nodes do not have child nodes anymore and their corresponding elements in the *length* and *f_pos* arrays can be reused for information on the points they index. The consolidation of *length*/*f_pos* for both quadtree nodes and points reduces memory footprint to a half. A running example is provided in Fig. 1, assuming that the predefined threshold representing the maximum number of points within a quadrant denoted as n_t is 12. Further assuming the root node has a level of 0, leaf node 2 is identified at level 1 (11 points), leaf nodes 4 and 7 are identified at level 2 (7 and 9 points, respectively) and leaf nodes 9, 10, 11, 12, 13 and 14 are identified at level 3.

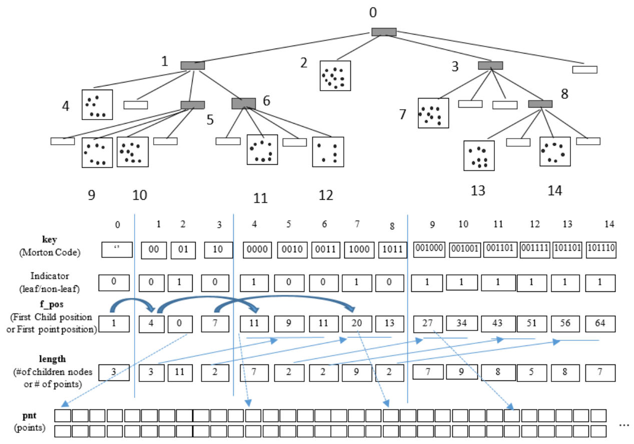

Fig. 1 SoA Data Layout of Quadtree on GPUs

Furthermore, by storing the positions of the **first** child nodes or the first points in quadrants, instead of the positions of **all** child nodes or points, the memory footprint can be significantly reduced. This is possible in our design as the quadtree nodes are laid out in a level-by-level manner and the positions of all child nodes under a parent quadtree node can be easily computed either sequentially or in parallel. The same argument can be applied to computing point positions where points among different quadrants are sorted based on the Morton codes of the quadrants they fall within. Note that points within a quadrant are unordered due to lack of practical needs, which also saves data movement costs in sorting. The SoA quadtree data layout will also help coalescing GPU memory accesses as neighboring threads in GPU thread blocks are assigned to process neighboring points or quadtree nodes.

We next briefly summarize in Section 3.2 the first phase of the top-down approach on identifying leaf quadrants which was published in [1] before introducing the new second phase on constructing the quadtree from leaf quadrants in Section 3.3. The top-down approach serves a baseline for comparison with our new bottom-up design and implementation introduced in Section 4.

3.2 Phase 1: Identifying Leaf Quadrants

The basic idea of the design of phase 1 in the top-down approach is similar to the previous works on GPU-based point indexing discussed in Section 2 in several aspects in the sense that they all adopt the top-down strategy that constructs quadtree nodes level-by-level. A running example shown in Fig. 2 provides an intuitive illustration. It can be seen that the indexing space is hierarchically partitioned and leaf nodes are identified in a top-down manner.

In the first step of Phase 1, Morton codes [7] of the input points are first generated in parallel by all available threads using a *transform* primitive where each thread applies a Z-order transformation [7] function to the point that is assigned to generate the output Morton codes. In the second step, the points to be indexed are then sorted based on points' Morton codes using the GPU-efficient radix sort (*stable_sort_by_key*). Third, the numbers of points in all valid quadrants are counted using a *reduce_by_key* primitive for a segmented reduction on sorted Morton codes which is also highly parallelizable where each thread is assigned to process a point. Different from the CUDA-based implementations in the previous works, both sort and segmented reduction are able to utilize all the threads and blocks in a GPU device (and potentially across multiple GPU devices and multiple computing nodes) for a large-scale point dataset where the number of points is typically far larger than the number of simultaneous threads of a GPU device.

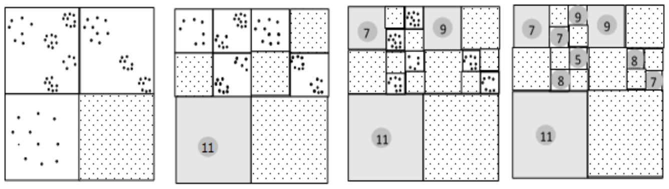

Fig. 2 Illustration of the Top-Down Quadtree Construction Approach

The next major step in phase 1 is to identify leaf quadrants that do not need further subdivisions and identify points that fall within these quadrants. The identified points are moved to the front of the point array so that the order of the last-level quadrants (leaf nodes) is the same as the order of the points that fall within them. Once this step is finished, the quadtree construction procedure can move to the next level.

During the identification process, similar to the previous works, quadrants that have fewer points than a predefined threshold are considered as leaf nodes in the quadtree and no further subdivisions are required. Conceptually, this is similar to a SQL statement like "SELECT * FROM Points WHERE #key IN (SELECT #key FROM Points GROUP BY #key HAVING COUNT (#key) < n_t)", here #key represents the Morton codes of quadrants. Due to space limit, we refer to [1] for the details of its implementation by chaining *copy_if*, *exclusive_scan*, *scatter_if*, *inclusive_scan* and *gather* primitives to expand the boolean input flag array derived from the sub-query of the SQL statement to an boolean output flag array to signal whether a point should be indexed by a leaf node at the present level or postponed to the next level.

Based on the boolean output flag array, in the final step of phase 1 (for each level), the points can be rearranged (or reordered) by using two *copy_if* primitives to copy the two types of points into a temporal point array and then use a *copy* primitive to copy the data back, all are highly parallelizable that can effectively use all GPU threads for large point datasets. Although device-to-device memory bandwidth on modern GPUs is in the order of hundreds of GB/s, similar to the CUDA sample code that utilizes double arrays, the extra memory footprint and data access cost are potentially sources of inefficiency, when compared with the newly proposed bottom-up approach (Section 4).

We note that rearranging points based on the separation between leaf and non-leaf quadtree nodes allows apply the same algorithm on the rest of the points (yet-to-be-indexed points) corresponding to the non-leaf quadtree nodes to construct next-level quadtree nodes. Upon the completion of Phase 1, among the four arrays of our SoA structure for quadtree, the Morton code array (*key*) and the leaf indicator array (*indicator*) have been filled level-by-level. Note that *indicator* array is filled by comparing the numbers of points in the resulting quadrants with the threshold value n_t after the *reduce_by_key* primitive. The *key* array is filled directly by the output of the key field after the *reduce_by_key* primitive.

3.3 Phase 2: Construction Quadtree from Leaf Quadrants

To complete the top-down quadtree construction technique, the remaining task is to fill the *length* array and the *f-pos* array. We have developed a new approach which is much simpler than the one presented in our earlier technical report [30] (not part of [1]). The overall process is provided in Fig. 3 with explanations to follow next. Note that "+=" is used to denote appending an array to an existing one and "~" denotes element-wise negation in an array. Arrays inside a pair of curly brackets form a zipped array, 0..*n* indicates a sequence, and a scalar number inside a pair of square bracket denotes lifting the scalar to an array. Finally, expression inside a pair of round brackets can be either parameters of a complicated functor (with implementation skipped) or the simple implementation of the functor for the corresponding primitive.

Assuming that the root node sits at level 0, starting from level 1 and for each level, line 3 computes the Morton codes for the parent nodes of all the nodes at the current level using a simple *transform* primitive with a functor that simply divides an input Mortan code by 4 (or right-shift 2 bits). Note that Morton codes of higher level quadrants have fewer bits. Line 4 uses a *reduce_by_key* primitive to count the numbers of child nodes of the parent nodes. Note that only non-leaf nodes that have one or more child nodes will appear in the resulting *p_key* array and *n_child* array, respectively.

```
Inputs: l_key: array of Morton Codes of leaf quadrants; n_point: Array of numbers of points in leaf quadrants; indicator: leaf/non-leaf indictor array; max_level
output: f_pos, length (Section 3.1)

Algorithm td_leafquad2tree
1  t-key ← l_key
2  for k=0, max_level-1
3      t_key ← transform(t_key) …(t_key[i]/=4)
4      (p_key, n_child)+=reduce_by_key(t_key)
5  n_map ← exclusive_scan (indicator)
6  length ← gather_if(n_point, n_map, indicator)
7  p_pos ← exclsive_scan(n_point)
8  f_pos ← gather_if(p_pos, n_map,indicator)
9  length ← copy_if(n_child, ~indicator)
10 n_child ← replace_if (n_child, indicator,0)
11 c_pos ← exclusive_scan(n_child)
12 f_pos ← copy_if(c_pos, ~indicator)
```

Fig. 3 Algorithm to Construct Quadtree from Leaf Nodes

The rest of the steps are to populate the *length* and *f_pos* arrays properly. Line 5 first accumulates the leaf node positions based on the *Indicator* array to generate a mapping array *n_map* using an *exclusive_scan* primitive. Line 6 puts *n_point* elements in the proper positions in the *length* array (for quadrants represented by leaf nodes) using a *gather_if* primitive and *n_map* and *indicator* as the inputs. Line 7 accumulates the first point positions for the quadrants represented by leaf nodes by using an *exclusive_scan* primitive again on *n_point*. A *gather_if* primitive is used in Line 8 in a way similar to Line 6.

The rest of the four lines populate the *length* and the *f_pos* arrays for non-leaf nodes. Line 9 uses a *copy_if* primitive and is based on the negates of the elements in the *indicator* array to populate the numbers of child nodes for non-leaf nodes in the length *array*. Line 10 sets the *n_child* elements to 0 for leaf nodes (using a *replace_if* primitive) to correctly compute the accumulated positions for non-leaf nodes in Line 11 using an *exclusive_scan* primitive. Finally, Line 12 populates the first child positions of non-leaf nodes based on the results of Line 11 using a *copy_if* primitive, which completes Phase 2 of the top-down approach.

Experiments show that the runtime of the Phase 2 code is insignificant comparing with that of Phase 1 (Section 5). This is expected as the numbers of the resulting quadrants and quadtree nodes are typically much smaller than the number of points, especially when the maximum number of points threshold is set to be relatively large (e.g., n_t=50). More details on the experiments are provided in Section 5.

4. THE NEWLY PROPOSED BOTTOM-UP APPROACH

4.1 Key ideas and Conceptual Design

While preliminary experiment results have shown that the top-down approach is capable of indexing ~170 million points of NYC taxi pick up locations in about a second on a RTX 2080 Ti GPU and is more efficient than the sample CUDA SDK code (Section 5), a question to ask is whether sorting yet-to-be-indexed points at all levels to maintain the correspondence between the BFS ordering of quadtree nodes and points they indexed is necessary, which turns to be the most expensive part of the top-down approach. Conceptually, since leaf quadtree nodes keep track of the points they index using the first point position values and the numbers of points that fall within the quadrants, by following the first child position values of intermediate quadtree nodes from the root to a leaf node, the points in the quadrants that satisfy the query criteria can be correctly retrieved. As such, it can be argued that the correspondence may not be essential, although it is typically the case for quadtree indexing approaches that follow a top-down strategy as in our previous work [1].

Without such a requirement, it turns out that points to be indexed need only be sorted once to generate quadtree nodes representing quadrants at the finest level based on their Morton codes (termed as full quadrants). Subsequently, a quadtree can be fully constructed from the full quadrants in a bottom-up manner with a few parallel primitives. By significantly reducing the workload on sorting points which is the bottleneck in our top-down approach, the overall performance of the newly proposed bottom-up approach can be significantly improved. We note that while the quadtrees constructed by the two approaches are identical, the orderings of the points that the quadtrees index may be different, as points in the two approaches are sorted differently. That is, the top-down approach sorts points based on Morton codes at multiple levels and the bottom-up approach sorts points based on Morton codes at the finest level.

It is worthy of noticing that, for the bottom-up approach, as points are sorted based on Morton codes, the order of points does not need to be changed for correct indexing when lower level quadrants are aggregated to upper level quadrants. Aggregation alone a space partitioning hierarchy will result in larger numbers of points (sum) and smaller first point positions (minimum) for upper level quadrants. The characteristics are fundamental to the correctness of the bottom-up approach while achieving efficiency. The running example in Fig. 4 provides an intuitive idea on the bottom-up approach where the lower level quadrants are aggregated bottom-up and the unqualified nodes are removed to construct a quadtree. We next present the design and implementation details of the newly proposed bottom-up approach.

4.2 Data Parallel Design and Primitive-based Implementation

The bottom-up approach starts with sorting points (using a *stable_sort_by_key* primitive) based on their Morton codes at the finest level (with the maximum depth) generated by a *transform* primitive. While both steps look similar to the steps discussed previously when presenting the top-down approach, we note that the top-down approach begins with the coarsest level Morton codes and the bottom-up approach begins with the finest level Morton codes. Subsequently, for the bottom-up approach, a *reduce_by_key* parallel primitive is applied to compute the Morton codes of the finest-level quadrants and count the numbers of points that fall within these quadrants.

Using these Morton codes and the counted numbers as the input arrays, in a way similar to Phase 2 in the top-down approach (Line 2-4 in Fig. 3 of Section 3), for each level of the quadtree, by using a *reduce_by_key* primitive, the Morton codes and the numbers of child quadtree nodes of the parent quadrant nodes can be computed. The process is repeated until reaching the root node. Although they share some similarities again, the difference is that, the inputs to the top-down and bottom approaches for this part are the leaf quadrants and the full quadrants, respectively.

We argue that the combined steps so far are conceptually equivalent to Phase 1 of the top-down approach. Different from the top-down approach that generates leaf quadrants with BFS order in Phase 1 which makes its Phase 2 much easier, Phase 1 of the bottom-up approach generates all possible non-empty quadrants (i.e., full quadrants) which makes its Phase 2 much more difficult. The rest of Section 4 is dedicated for Phase 2 of the bottom-up approach. As a summary, after the above three steps (Phase 1), the outputs have four arrays which also serve as the input for Phase 2: an array of Morton codes of quadrants (***pkey***), an array of the numbers of non-empty sub-quadrants (***clen***), an array of the numbers of points in these quadrants (***nlen***), and finally, an array of sorted points based on the Morton codes of the quadrants at the last level (***pnt***), where *pkey*, *clen* and *nlen* have the same lengths and their elements have a one-to-one correspondence.

Two major issues remain in Phase 2. First, as shown in Fig. 4, it can be seen that, among the full quadrants, some cannot be represented as valid quadtree nodes and should be removed (more details shortly). Second, although the *key* and the *length* arrays are filled during the bottom-up level-by-level iterations, the *f_pos* array and the *indicator* array need to be computed. The algorithm to tackle these two issues and generate a quadtree from full quadrants are provided in Fig. 5 and its details are provided in the two subsections to follow next.

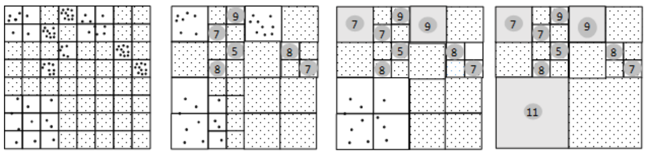

Fig. 4 Illustration of the Bottom-Up Quadtree Construction Approach

```
Inputs: pkey: array of Morton codes of quadrants; clen:
numbers of non-empty sub-quadrants; nlen: array of the
numbers of points in these quadrants; n_t
Output: indicator, f_pos

Algorithm genValidQuadrants
1 tpos←exclusive_scan(clen)
2 tmap←scatter ([0..|clen|],tpos)
3 tmap←inclusive_scan(tmap, maximum)
4 tlen←clen
5{pkey,clen,nlen,tmap}←remove_if({pkey,clen,
nlen,tmap}, (nlen, n_t))
6 indicator←transform(clen,( n_t ))
7 nlen←replace_if(nlen, indicator)
8 ppos ←exclusive_scan(nlen)
9 clen←replace_if(clen, ~ indicator)
10 cpos ←exclusive_scan(clen)
11 f_pos ←transform({ppos,cpos}, indicator)
```

Fig. 5 Bottom-Up Quadtree Construction Algorithm from Full Quadrants

4.2.1 Identifying Valid Quadtree Nodes

Recall that, in the top-down approach, leaf nodes can be simply identified by comparing the number of points that fall within the quadrants, i.e., n_k, with the threshold value n_t. As points that belong to leaf nodes and non-leaf nodes are re-ordered in the output point array at each level, the positions of the first points in the leaf nodes are simply pre-fix sums (exclusive scan) of the numbers of points in these quadrants. As discussed in Section 3.1, points under the quadrants represented by non-leaf nodes need to be re-ordered and sorted based on their Morton codes at each level to maintain the order, which is the major bottleneck of the top-down approach.

For the bottom-up approach, it can be seen that whether a quadrant is represented by a leaf node or non-leaf node depends not only on n_k but also on the number of points that fall within its **parent** node (n_p). The nodes corresponding to the last level quadrants with $n_k > n_t$ automatically qualify as leaf nodes. However, for the rest of the quadrants, only when $n_k <= n_t < n_p$ can the node be qualified as a valid leaf node. Otherwise, the quadrants need to be aggregated until a valid leaf node can be identified. Nodes representing such lower level quadrants are not part of the final quadtree and must be removed. As shown in Fig. 4, all the quadrants covered by the leaf node with 11 points (the lower-left part of the four sub-panels, highlighted) should be removed and only the leaf node should appear in the final constructed quadtree. The problem can be trivially solved if we keep a pointer (or array offset) to the parent node of the current node but our SoA quadtree data structure has only the *length* and the *f_pos* arrays for the memory efficiency reason (Section 3.1 and Fig. 1).

To construct a quadtree with only the numbers of points in the last-level quadrant and the numbers of child nodes for all quadrants derived from the last-level quadrants, our approach transforms the criteria for identifying leaf-nodes into new ones. The new criteria can be verified in two successive steps where each step can be implemented by one or more parallel primitives. The new criteria are that, first, if a quadrant whose **parent** quadrant has no more than n_t points, i.e., $n_p <= n_t$, it should be removed. For the remaining quadrants, if $n_k <= n_t$, they should be represented by leaf nodes; otherwise, they should be represented by non-leaf nodes. The algorithm with the primitive based implementation is illustrated in the first five lines of Fig. 5.

Lines 1-3 compute the numbers of points in the quadrants' parent quadrants, given the numbers of points of child quadrants under these parent quadrants. Essentially this is equivalent to computing the offsets of parent quadrants for all child quadrants on-the-fly as the numbers of points and their quadrants have a one-to-one correspondence in their respective arrays as part of the input. At an abstract level, this can be realized by a special *expand* parallel primitive which is a chain of *exclusive_scan - scatter - inclusive_scan*. The *exclusive_scan* primitive takes the default "*plus*" as the functor parameter to accumulate the numbers of children into the positions of the first child nodes as we have discussed before. The *scatter* primitive puts the sequence numbers of the parent nodes into these first child node positions. Finally, the *inclusive_scan* primitive with "*maximum*" as the functor fills in the blanks between the consecutive first node positions with the sequence identifier of the first child node. This is because of the fact that all child nodes have the same sequence identifiers of their parent nodes. The compound *expand* primitive is special for two reasons. First, as the numbers of child nodes of a parent node are always greater than 0 and thus *scatter*, instead of the conditional *scatter_if*, can be used, which is more efficient. Second, as the sequence identifiers of parent quadtree nodes, which are sequential and range from 0 to |*clen*|, are exactly what we want to expand, the last step with a *gather* primitive to actually replicate the input data items (especially when these data items are not comparable) can be omitted.

Line 5 is the key step where the unqualified quadrants are removed by using *a remove_if* primitive. To keep the correspondence among *qkey*, *clen* and *nlen* arrays, we zip them together as a single input vector for the primitive. This brings a caveat that needs special attention, due to the reason that *remove_if* is an in-place primitive. When the elements of the input array are applied to the *remove_if* primitive in parallel, they are being modified during the primitive's execution. Unfortunately, the primitives' functor also rely on *nlen* to compare its elements with n_t where we expect *nlen* array to be constant, which results in a semantic conflict. The issue is solved by duplicating the *nlen* array and use its copy for the functor (Line 4). The duplicate is then deleted after its lifecycle is over to reduce GPU memory footprint.

4.2.2 Populating INDICATOR and F_POS arrays

The leaf and non-leaf *indicator* array can be populated by applying the second criteria discussed previously to the output of Phase 1 to decide whether a quadtree node above the last level is a leaf node or not. This is implemented by a *transform* primitive using a simple comparison functor by comparing the numbers of points under the quadrants with the threshold n_t as shown in Line 6 of Fig. 5. The elements are set to non-leaf when $n_k>n_t$. Line 7 takes care of the quadrants at the last level where they are considered as leaf nodes even if $n_k>n_t$. This can be simply implemented by a *transform* primitive to set the respective elements in the *indicator* array to true.

The last five lines in Fig. 5 are used to fill the *length* array in the quadtree data structure based on *clen*, *nlen* and the *indicator* arrays. Line 8 changes the elements in the *nlen* array to 0 for non-leaf quadtree nodes by a *replace_if* primitive before applying an *exclusive_scan* primitive to accumulate the *nlen* array into a *ppos* array to record the offsets of the first point positions at Line 9. Note that the *nlen* array elements with zero values (for non-leaf nodes) do not increase the corresponding *ppos* values and these values are irrelevant to the final *length* array. Similarly, the *clen* array are accumulated into the *cpos* array after setting the *clen* elements corresponding to the leaf nodes to 0 in Line 10 and Line 11, respectively. Again, the *clen* elements with zero values do not increase the corresponding *cpos* elements. Finally, step 11 assembles the *ppos* and *cpos* arrays into the first position array *f_pos* based on the leaf *indicator* array, which is naturally implemented as a *transform* primitive using a simple switch function as the functor with *ppos* and *cpos* as the inputs. Using classic C syntax, the functor can be expressed as *f_pos*[i]=(*indicator*[i])?*ppos*[i]:*cpos*[i] for easy interpretation.

5. EXPERIMENTS & RESULTS

5.1 Data and Experiment Setup

Among the four top-down approaches discussed in details by the end of Section 2, only the CUDA SDK sample code is publically available. We thus compare the end-to-end runtimes produced by the CUDA SDK sample code (or simply SDK), our top-down approach (TD), and our bottom-up approach (BU). However, the SDK code's memory allocation scheme prevents it from constructing large quadtrees with the maximum level/depth larger than 14 on typical GPUs (e.g., RTX 2080 Ti with 11 GB memory and Titan V with 12 GB memory). As such, *max_level* in the SDK code is set to 14. For the top-down and bottom-up approaches that we have developed, the memory footprints are generally linear with respect to the numbers of points to index and not directly related to the maximum depth/level limits. Using a *max_level* =16 is already capable of indexing points in a space of $2^{16}*2^{16}$. For resolution as high as 1 meter, which is much higher than typical GPS location accuracy around 30 meters, the index space is about 65*65 kilometers, which should be sufficient for most city scale applications. As such, we set the maximum depth/level to 16 in our top-down and bottom-up approaches. The maximum number of points in a leaf quadrant (except for the last level) n_t is set to 200.

While it would be also interesting to use synthetic data with random distribution or some skewed distributions, we are more interested in the performance on real world data that are typically unevenly distributed and difficult to be approximated by simple mathematical distributions. We have picked a popular dataset which is the taxi trip pickup/drop-off locations in New York City (NYC) with yearly ~170 million points [31] for experiments. In this study, we have chosen the pickup locations in 2009 which has 168,898,952 points. The original latitude/longitude coordinates have been re-projected into the standard EPSG 2263 projection that is typically adopted for city-level geospatial applications in NYC and its neighboring Long Island area [32]. The unit of the projected coordinates is foot which is suitable for direct distance computation without further processing. To test the scalability of the designs and the implementations of the approaches, we accumulate the first 1-12 months' pickup location data and treat them as 12 datasets, each for an experiment.

All experiments are performed on a Nvidia GTX 2080 Ti GPU with 4,325 CUDA cores running at 1.65 GZ and 11 GB GDDR5 memory with 352-bit memory bandwidth. All implementations are compiled with CUDA SDK version 10.1, computing capability 7.5 and -O3 optimization flag. We measured the maximum memory footprints for the top-down and the bottom-up approaches, which are 5.99GB and 3.15GB, respectively. Although we leave fine-tuning memory management for future work, e.g., reusing temporal arrays and finer-grained memory allocation/deallocation, the current implementations can run on inexpensive commodity GPUs with 8GB memory, which is becoming the mainstream for the current generation of Nvidia GPUs. As the bottom-up approach requires only about half of the GPU memory of the top-down approach, it is suitable to run on even lower-end GPUs with as little as 4GB memory for the yearly NYC taxi trip data.

5.2 Results

The numbers of points and the end-to-end runtimes of the three approaches for the 12 experiments are listed in Table 1. The last two columns of Table 1 also list the speedup of our top-down implementation over the CUDA SDK sample code and the speedup of the bottom-up approach over the top-down approach. It can be seen that the top-down approach (*max_level*=16) is about 2.5X faster than the CUDA SDK sample code (*max_level*=14) while the bottom-up approach is 3.4X faster than the top-down approach. The speedups are quite consistent across the 12 experiments for both comparisons. This is likely due to the similar distributions of taxi pickup locations in NYC across different months (and likely across multiple years). Due to the inferior performance of the CUDA SDK sample code, we exclude it from further discussion. As a summary, the bottom-up approach not only runs faster (3.4X) but also is more memory efficient (~2X), when compared with the top-down approach.

Table 1 Runtimes of Three Approaches (in milliseconds): CUDA SDK Sample Code (SDK), Top-Down (TD) and Bottom-up (BU) and Speedups

# Mo.	#of points	SDK (T1)	TD (T2)	BU (T3)	Speedup =T1/T2	Speedup =T2/T3
1	13,887,620	360.4	143.3	40.8	2.52	3.51
2	27,079,723	619.4	244.4	68.5	2.53	3.57
3	41,284,081	833.4	336.7	98.6	2.48	3.41
4	55,383,596	1057.1	426.7	128.8	2.48	3.31
5	69,970,743	1290.6	535.5	150.1	2.41	3.57
6	84,035,490	1549.0	608.8	189.3	2.54	3.22
7	97,553,533	1769.2	711.5	199.1	2.49	3.57
8	111,127,610	1977.6	786.8	226.4	2.51	3.48
9	124,993,700	2206.7	869.3	253.9	2.54	3.42
10	140,444,141	2463.6	966.0	287.8	2.55	3.36
11	154,523,740	2685.2	1050.0	314.5	2.56	3.34
12	168,898,952	2959.5	1124.9	339.1	2.63	3.32

To further understand the performance differences between the top-down and the bottom-up approach, we have listed the breakdown times of the three components in both approaches, i.e., the initialization time, Phase 1 time and Phase 2 time. The initialization part is responsible for GPU memory allocation and CPU to GPU data transfer; the initialization times are listed as TD-I and BU-I in Table 2 for the two approaches, respectively. Although the top-down and the bottom-up approaches utilize slightly different data structures, the runtime on transferring point data from CPU to GPU which is common to both approaches, dominates both TD-I and BU-I. As a result, TD-I and BU-I are very close. For situations that point data is already on GPU devices, TD-I and BU-I will be close to 0 and can be excluded from their respective runtimes.

Table 2 Breakdown Runtimes (in milliseconds) of the Top-Down Approach (TD) and Bottom-Up Approach

# Mo.	TD-I	TD-P1	TD-P2	BU-I	BU-P1	BU-P2	Speedup
1	11.9	119.9	11.5	11.9	13.4	15.5	4.55
2	22.7	207.2	14.5	22.8	22.2	23.4	4.86
3	34.3	287.2	15.2	34.7	31.7	32.2	4.73
4	45.8	365.9	14.9	45.9	41.2	41.7	4.59
5	58.4	456.1	21.0	57.5	52.0	40.6	5.15
6	69.6	517.2	22.0	69.7	61.4	58.2	4.51
7	80.3	609.3	21.8	80.2	71.6	47.3	5.31
8	91.4	674.0	21.5	91.6	80.3	54.6	5.16
9	102.3	744.4	22.6	104.8	89.7	59.4	5.14
10	115.2	829.3	21.5	115.8	96.4	75.7	4.94
11	126.8	900.8	22.4	128.3	111.7	74.5	4.96
12	138.6	965.4	20.9	139.6	120.7	78.9	4.94

I: Initialization time – GPU memory allocation and CPU->GPU data transfer
P1 and P2: phase 1 and phase 2
Speedup=(TD-P1+TD-P2)/(BU-P1+BU-P2)

From Table 2, it can be seen that the runtime for Phase 1 (TD-P1) in the top-down approach is much larger than that of Phase 2 (TD-P2) as we have discussed in the previous sections. In fact, the difference between TD-P1 and TD-P2 gets larger as the numbers of points increase. As a matter of fact, TD-P1 over TD-P2 increases from 10.4X (119.9/11.9) to 46.2X (965.4/20.9) from 1 month to 12 months. When comparing TD-P1 and TD-P2 for 1 month and 12 months, TD-P1 increases 8.1X while TD-P2 increases only 1.8X. This is because the resulting numbers of leaf quadrants which are the inputs of TD's Phase 1, grow sub-linearly with respect to the numbers of points, which are the input for Phase 2. As a result, it is likely that, as the number of points to index increases, the resulting leaf nodes become more full, but the numbers of points in these quadrants are still less than the threshold n_t. When comparing BU-P1 and BU-P2, we can see that they are much closer, although BU-P2 is still lower than BU-P1 for all months.

The results of Table 2 also support our previous discussions that Phase 1 in the top-down approach is more complex than Phase 1 in the bottom-up approach due to the fact that TD-P1 sorts yet-to-be indexed points at each and every level while BU-P2 sorts all points to be indexed only once at the finest level. In contrast, TD-P2 is much simpler than BU-P2 as TD-P2 constructs a quadtree from leaf quadrants only while BU-P2 constructs a quadtree from full quadrants. In addition to the reason that the number of full quadrants can be several times larger than the number of leaf quadrants, BU-P2 also needs sophisticated logic to remove quadrants that cannot be qualified as leaf quadtree nodes among the full quadrants.

Assuming that the point data to be indexed are already on GPU devices, the total runtime of the top-down approach (TD-tot) would be just TD-P1+TD-P2 and the total runtime of the bottom-up approach (BU-tot) would be just BU-P1+BU-P2. The speedup, defined as TD-tot/BU-tot, is computed. As shown in the last column of Table 2, the speedup is about 4.9X across the 12 experiments, which is higher than 3.4X when the initialization time (mostly CPU to GPU data transfer time) is included.

Although the exact runtimes of quadtree constructions reported in [28] are unavailable, it can be seen from its Fig. 8 that it takes about 21 milliseconds to index 9.5 million points for skewed distribution, which is already the largest in the experiment that generates the figure. Our bottom-up approach indexes 13.9 million points (the first month in the NYC taxi trip dataset) in about 28.9 milliseconds, which suggests slightly higher performance of our bottom-up approach, i.e., 481 million points/s vs. 452 million points/s, for relatively small-scale data. Since Fig. 8 of [28] exhibits super-liner increase of runtime with respect to the numbers of points for skewed data, extrapolating the runtime to 169 million points would not be accurate or even possible. Nevertheless, even assuming a linear extrapolation for [28], the achieved performance of our bottom-up approach, i.e., indexing 169 million points in 199.6 ms which is equivalent to 846 million points per second, would still perform about 2X faster.

Overall, the total runtime of the proposed bottom-up approach is capable of indexing approximately 170 million points in about 200ms. With an indexing rate of 850 million points per second, it may suggest the possibility of real-time and on-the-fly indexing for point datasets at scale, even on commodity GPUs that cost around $1200 or less. The high-performance may open opportunities for interactive explorations through the emerging GPU-based data management systems such as Nvidia cuDF [2], as discussed in the introduction section. We are in the process of integrating the quadtree indexing approach as well as several other GPU-based spatial query techniques into cuDF.

6. CONCLUSION AND FUTURE WORK

In this study, after extending our previous work on identifying leaf quadrants from a large-scale point dataset by repetitively partitioning space into quadrants until the numbers of points in each quadrant is smaller than a predefined threshold for parallelizing spatial joins to a full quadtree indexing approach, termed as the top-down approach, we have developed a new and more efficient bottom-up approach. While not exactly, the top-down and the bottom-up approaches share some duality features in indexing point data which makes exploring the two approaches simultaneously interesting. We present the design and implementation details of the extension of the top-down approach and the complete bottom-up approach. Different from previous works whose implementations adopt plain CUDA programming, our implementations are based on parallel primitives which are simple to understand, easy to implement, and offer high level of portability.

Experiments on the yearly 170 million taxi pickup locations in NYC in 2009 have shown that the top-down implementation is about 2.5X faster than the quadtree construction code that is shipped as a CUDA SDK sample even though our top-down implementation indexes with a maximum level/depth of 16 while the SDK code is only capable of indexing with a maximum level/depth of 14 before running out of memory. Both running at a maximum level/depth of 16, the bottom-up approach is 3.4X better than the top-down approach when CPU to GPU data transfer time is included. The speedup increases to 4.9X when the point data to be indexed are already on GPU devices and the data transfer time is not needed. With an indexing time of about 200 milliseconds to index about 170 million points, our bottom-up approach seems to be capable of real-time and on-the-fly indexing for interactive explorations, even on inexpensive commodity GPUs.

For future work, first, as discussed inline, we would like to fine-tune the memory management part to reduce both memory allocation/deallocation time and to minimize memory footprint. The RAPIDS Memory Manager (rmm [33]), which is also used in cuDF, can be a good option for the more efficient memory allocation/deallocation on GPUs. Carefully analyzing the lifecycle of each temporal variables and deallocating (or offloading to CPU memory) promptly on variables that are no longer needed or cannot be reused seem to be a low hanging fruit for this purpose. The task could be easier for data parallel designs and primitive-based implementations as most of their important variables are arrays and the number of such variables tends to be small. Second, we plan to integrate the indexing techniques into GPU-based data management systems such as Nvidia cuDF (part of RAPIDS) to extend such systems to manage both relational and spatial (and spatiotemporal/trajectory) data with indexing support. There are considerable software engineering challenges to make such a system available and we plan to tackle them.

ACKNOWLEDGEMENT

This work is supported through NSF Grants IIS-1302423 and IIS-1302439 and PSC-CUNY 60777-00 48. Part of the work is done while Jianting Zhang is visiting Nvidia Corporation.

7. REFERENCES

[1] J. Zhang and S. You:, "Speeding up large-scale point-in-polygon test based spatial join on GPUs.," in *Proc. ACM SIGSPATIAL BigSPatial Workshop*, 2012.

[2] Nvidia, "cuDF - GPU DataFrames," [Online]. Available: https://github.com/rapidsai/cudf.

[3] S. Breß, H. Funke and J. Teubner:, "Robust Query Processing in Co-Processor-accelerated Databases," Proc. ACM SIGMOD, 2016.

[4] S. Zhang, J. He, B. He and M. Lu, "OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1374--1377, 2013.

[5] J. Zhang, S. You and L. Gruenwald, "Large-Scale Spatial Data Processing on GPUs and GPU-Accelerated Clusters," *ACM SIGSPATIAL Special,* vol. 6, no. 3, pp. 27-34, 2014.

[6] S. K. Prasad, M. McDermott, S. Puri, D. Shah, D. Aghajarian, S. Shekhar and X. Zhou, "A vision for GPU-accelerated parallel computation on geo-spatial datasets," *ACM SIGSPATIAL Special,* vol. 6, no. 3, pp. 19-26, 2014.

[7] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2006.

[8] D. A. lcantara, Sharf, rei, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens and N. Amenta, "Real-time Parallel Hashing on the GPU," *ACM Trans. Graph.,* 28(5), #154, 2009.

[9] M. Nießner, M. Zollhöfer and M. Stamminger, "Real-time 3D reconstruction at scale using voxel hashing," *ACM Trans. Graph.,* vol. 32, no. 6, #169, 2013.

[10] J. Zhang, S. You and L. Gruenwald:, "Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs.," *Information Systems,* vol. 44, pp. 134-154 , 2014.

[11] D. Aghajarian, S. Puri and S. K. Prasad:, "CMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform," in *SIGSPATIAL/GIS 2016*, 2016.

[12] S. You, J. Zhang and L. Gruenwald, "High-Performance Polyline Intersection based Spatial Join on GPU-Accelerated Clusters," in *Proc. ACM SIGSPATIAL BigSpatial Workshop*, 2016.

[13] M. McCool, A. Robison and J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.

[14] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 2nd ed., Morgan Kaufmann, 2012.

[15] Nvidia, "Thrust Parallel Library," [Online]. Available: https://thrust.github.io/.

[16] M. B. Brahim, W. Drira, F. Filali and N. Hamdi, "Spatial data extension for Cassandra NoSQL database," *Journal of Big Data,* vol. 3, no. 11, 2016.

[17] J. Zhang and S. You:, "High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives," *International Journal of Geographical Information Science* , vol. 27, no. 11, pp. 2207-2226, 2013.

[18] J. Zhang, S. You and L. Gruenwald:, "Parallel Selectivity Estimation for Optimizing Multidimensional Spatial Join Processing on GPUs.," in *Proc. ICDE HardDB Workshop*, 2017.

[19] J. Zhang, S. You and L. Gruenwald, "Data Parallel Quadtree Indexing and Spatial Query Processing of Complex Polygon Data on GPUs," in *Proc. ADMS@VLDB*, 2014.

[20] N. Satish, M. J. Harris and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. IPDPS*, 2009.

[21] D. Merrill and A. S. Grimshaw:, "High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing," *Parallel Processing Letters,* vol. 21, no. 2, pp. 245-272, 2011.

[22] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum,* vol. 28, no. 2, pp. 375-384, 2009.

[23] K. Zhou, Q. Hou, R. Wang and B. Guo, "Real-time KD-tree Construction on Graphics Hardware," *ACM Trans. Graph.,* vol. 27, no. 5, pp. 126:1--126:11, 2008.

[24] F. Gieseke, J. Heinermann, C. Oancea and C. Igel, "Buffer K-d Trees: Processing Massive Nearest Neighbor Queries on GPUs," in *Proc. ICML*, 2014.

[25] M. Kelly and A. Breslow, "Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach," 2011.

[26] J. Gluck and A. Danner, "Fast GPGPU Based Quadtree Construction," 2014.

[27] Nvidia, "Quad Tree Construction," [Online]. Available: https://github.com/huoyao/cudasdk/tree/master/6_Advanced/cdpQuadtree.

[28] Z. Nouri and Y.-C. Tu, "GPU-based parallel indexing for concurrent spatial query processing," in *Proc. SSDBM*, 2018.

[29] Nvidia, "CUDA C Programming Guide," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] J. Zhang and L. Gruenwald, "Spatial Indexing of Large-Scale Geo-Referenced Point Data on GPGPUs," Technical Report, 2012. [Online]. Available: http://geoteci.engr.ccny.cuny.edu/primcsptp/CSPTP_tr.pdf

[31] NYC TLC, "TLC Trip Record Data," [Online]. Available: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[32] EPSG, "EPSG:2263 Projection," [Online]. Available: https://epsg.io/2263.

[33] Nvidia, "RMM: RAPIDS Memory Manager," [Online]. Available: https://github.com/rapidsai/rmm.

Appendix: A Quick Reference of Thrust Parallel Primitives

(1) *Sort, sort_by_key* and *stable_sort_by_key*.
https://thrust.github.io/doc/group__sorting.html

While *sort* performs a key-only sort, *sort_by_key* also takes a value array and performs a key-value sort. *Stable_sort_by_key* preserves the relative ordering of equivalent elements which is more expensive than *sort_by_key* but may be desirable in certain applications.

(2) *Reduce* and *reduce_by_key*.
https://thrust.github.io/doc/group__reductions.html

Reduce is used to accumulate a vector array to a scalar value. For example, reduce([3,2,4])→11. While the summation (using a default "plus" functor) is frequently used in reductions, Thrust allows using a user defined associative binary function for tailored summation, such as determining the maximum entry ("maximum" functor) or computing bounding boxes of points (useful to have an additional bbox array in the quadtree SOA structure). *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 ([1,3,3,2],[2,1,3,4])→([1,3,2],[2,4,6]).

(3) *Scan* and *scan_by_key*.
https://thrust.github.io/doc/scan_8h.html

The *Scan* primitive computes the cumulative sums 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* works as ([3,2,4])→([0,3,5]) while *inclusive_scan* works as ([3,2,4])→([3,5,9]). Similarly, *scan_by_key* works on consecutive key groups instead of a whole vector/array. In this research, *inclusive_scan_by_key* and *exclusive_scan_by_key* are extensively used to compute the positions of entries in a vector after applying *reduce_by_key* which outputs numbers of entries with same keys.

(4) *Copy* and *copy_if*.

https://thrust.github.io/doc/group__copying.html

https://thrust.github.io/doc/group__stream__compaction.html

The functionality of the two primitives is self-evident by names. In this research, we use *copy* to move groups of entries from one location to another, mostly within a same vector. The conditional *copy_if* primitive is mostly used for identifying points and keys that satisfy certain criteria expressed as a boolean array and output the identified entries to a new vector for further processing.

(5) *Remove_if*.

https://thrust.github.io/doc/group__stream__compaction.html

Remove_if marks elements in a vector that satisfy a predicate and compact the unmarked elements to the beginning of the vector so that the marked elements are removed. For example, *Remove_if* works as ([1, 4, 2, 8, 5, 7,is_even])→[1,5,7]. *Remove_if* is functionally equivalent to *copy_if* but it allows in-place operation in the Thrust library. In contrast, using *copy_if* would require a temporary vector and *remove_if* is more convenient in this case.

(6) *Transform*.
https://thrust.github.io/doc/group__transformations.html

The basic form of *Transform* applies a unary function to each entry of an input sequence and stores the result in the corresponding position in an output sequence. *Transform* is more general than *copy* as it allows a user defined operation (functor) to be applied to entries rather than simply copying. The functor can be reasonably complex as long as each and every element in the input array is expected to apply the same logic in the functor. The functor can also takes global memory pointers as its input parameter so that the functor can access additional data besides its input element from the input array.

(7) *Gather, Gather_if, Scatter and Scatter_if*.

https://thrust.github.io/doc/group__gathering.html

https://thrust.github.io/doc/group__scattering.html

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],[*,*,*,*,*,*])→([4,*,8,*12,*]). Similar to *copy_if* and *remove_if*, *gather_if* and *scatter_if* take an additional boolean sign array and perform gather/scatter on the input elements only when the corresponding sign values are true.