GPU-Based Parallel Indexing for Concurrent Spatial Query Processing

Zhila Nouri and Yi-Cheng Tu
University of South Florida
4202 E. Fowler Ave., ENB 118
Tampa, Florida 33620
{zhila,tuy}@mail.usf.edu

ABSTRACT

In most spatial database applications, the input data is very large. Previous work has shown the importance of using spatial indexing and parallel computing to speed up such tasks. In recent years, GPUs have become a mainstream platform for massively parallel data processing. On the other hand, due to the complex hardware architecture and programming model, developing programs optimized towards high performance on GPUs is non-trivial, and traditional wisdom geared towards CPU implementations is often found to be ineffective. Recent work on GPU-based spatial indexing focused on parallelizing one individual query at a time. In this paper, we argue that current one-query-at-a-time approach has low work efficiency and cannot make good use of GPU resources. To address such challenges, we present a framework named G-PICS for parallel processing of large number of concurrent spatial queries over big datasets on GPUs. G-PICS is motivated by the fact that many spatial query processing applications are busy systems in which a large number of queries arrive per unit of time. G-PICS encapsulates an efficient parallel algorithm for constructing spatial trees on GPUs and supports major spatial query types such as spatial point search, range search, within-distance search, k-nearest neighbors, and spatial joins. While support for dynamic data inputs missing from existing work, G-PICS provides an efficient parallel update procedure on GPUs. With the query processing, tree construction, and update procedure introduced, G-PICS shows great performance boosts over best-known parallel GPU and parallel CPU-based spatial processing systems.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*;

ACM Reference format:
Zhila Nouri and Yi-Cheng Tu. 2018. GPU-Based Parallel Indexing for Concurrent Spatial Query Processing. In *Proceedings of 30th International Conference on Scientific and Statistical Database Management, Bozen-Bolzano, Italy, July 9–11, 2018 (SSDBM '18),* 12 pages.
https://doi.org/10.1145/3221269.3221296

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SSDBM '18, July 9–11, 2018, Bozen-Bolzano, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6505-5/18/07…$15.00
https://doi.org/10.1145/3221269.3221296

1 INTRODUCTION

Over the past few years, there has been great interest towards spatial query processing motivated by increasing availability of spatio-temporal data. There are a large number of applications such as geographic information systems (GIS), mobile computing, scientific computing, epidemic simulation, astrophysics, just to name a few. Popular spatial queries include spatial point search, range search, within-distance search, k-nearest neighbor (kNN), and spatial joins [23, 36]. Index-based access methods, which reduce the number of points to be searched [32], are shown to be the prerequisite of having efficient spatial query processing in large datasets. Parallelization is another common approach in achieving high performance while dealing with spatial databases. Previous work demonstrated the great potential of parallel algorithms in query processing [7, 11, 16, 37, 38]. On the other hand, if parallelism is adopted without spatial data structures in query processing, the performance gain obtained will fade away quickly as data size increases [6, 9, 14, 35].

Over the last decade, many-core hardware has been adapted to speed up high-performance computing (HPC) applications [4]. Among them, Graphical Processing Units (GPUs) have become a mainstream platform for massively parallel data processing [26]. A modern GPU usually has thousands of thin computational cores that are organized into an array of streaming multiprocessors (SMs). There are also several layers of memory with different accessibility, and accessing costs. First, there are multiple gigabytes of off-chip *Global Memory* to which all the SMs have read and write accesses simultaneously, and has the largest capacity on GPUs. Second, each SM has programmable high-speed shared memory and non-programmable L1/L2 cache for local memory accesses. Considering the software side for programming, thousands of fine-grained threads can be launched simultaneously. Due to the complex hardware architecture and programming model, developing programs optimized towards achieving high performance on GPUs is non-trivial and traditional wisdom geared towards (even multi-core) CPU implementations is often found to be ineffective.

Recent work [12] presented a GPU-based spatial index called STIG (Spatio-Temporal Indexing using GPUs) to support spatio-temporal search queries [12]. Based on the k-d tree, a STIG tree consists of intermediate nodes, and a set of leaf blocks to store the data records in consecutive memory locations. A STIG tree is constructed using a serial algorithm on the CPU, and then transferred to the GPU for query processing. Spatial query execution over STIG (Figure 1) consists of two steps : (1) leaf nodes satisfying the search conditions are identified; and (2) all data points in the identified nodes are examined to determine the final results. STIG processes

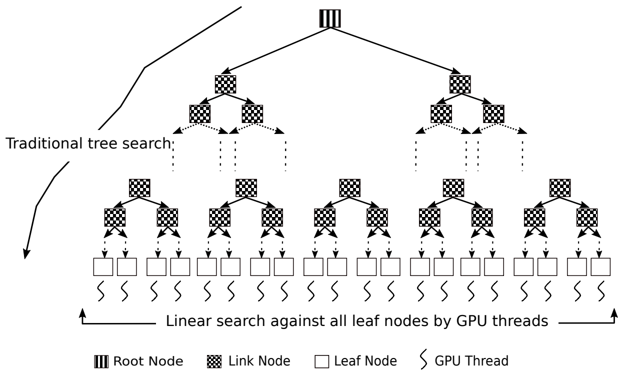

Figure 1: Traditional tree search versus parallel linear search against all leaf nodes in STIG

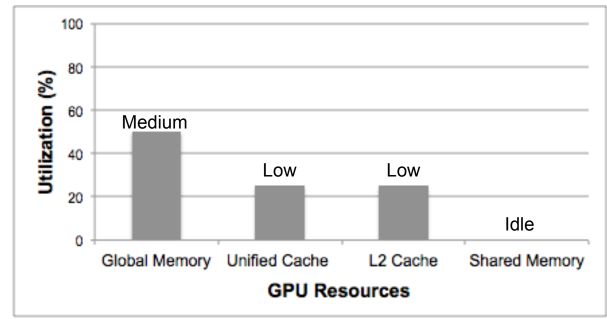

Figure 2: GPU resource utilization in running a STIG-like kernel using Nvidia Tesla P100

one query at a time. However, in parallelizing one tree search, it is not easy to achieve a high degree of parallelism, especially when the algorithm visits higher levels of the tree. Hence, STIG adapts a data parallel solution for this step on GPU in which all the leaf blocks are transferred to GPU and scanned in a brute-force manner. This idea was followed by Chavan *et al.* [8] for spatio-temporal data visualization. In this work, a variation of the quadtree was used for indexing the input data. Similar to STIG, the tree structure is built on CPU, and leaf nodes with their attached data records are transferred to GPU for query processing.

We argue that the STIG approach, which scans all leaf nodes, is fundamentally inefficient as it literally changes the total amount of work (*work efficiency*) from logarithmic to linear (Figure 1). Although STIG takes advantage of the thousands of GPU cores to process leaf nodes concurrently, the speedup can be quickly offset by the growing number of leaf nodes in large datasets. In this paper, we propose an approach that bears the logarithmic work efficiency for each query yet overcomes the problem of low parallelism level. Instead of parallelizing a single tree-search operation, our strategy is to **parallelize multiple queries running concurrently**. It is well-known that many location-based applications are busy systems with very high concurrent query arrival rate [25, 30]. Moreover, in scientific simulations such as molecular and astrophysical simulations, millions of spatial queries such as kNNs and range searches are issued at every step of the simulation [19]. The batch query processing approach in our solution achieves task parallelism on GPUs, allowing each thread to run an individual query. A search query can therefore be done in logarithmic steps. Because each thread roughly carries the same work and is independent to others, there is no problem in parallelizing the many tree search operations.

Another advantage of our approach is we achieve much better GPU resource utilization in the second step of spatial query processing, i.e., retrieving query results from the leaf nodes. Within the GPU device, global memory has the highest access latency. In STIG, Each query scans a list of leaf nodes to find their data records. Therefore, the same data record can be accessed many times by different queries in a workload. As a result, the program easily hits a performance ceiling due to congestion of global memory. To show this, we implemented STIG following the descriptions in [12]. Figure 2 shows the utilization of multiple GPU resources recorded by NVidia Visual Profiler (NVVP) for a within-distance search query against a 16-million-point dataset. Clearly, most of the accesses are fulfilled by global memory while other high performance resources are either insufficiently utilized or largely unused (e.g., shared memory).

Finally, an important feature that is missing from existing work is the support of updates. In STIG, the tree is constructed in host memory and transferred to GPU global memory. In large datasets, building a tree is costly, and cost of transferring data from CPU to GPU is significant. The CPU communicates with GPU's global memory through a PCI-E bus which is of much lower bandwidth and higher latency as compared to the global memory. For static data, it is not an essential issue as tree construction and transferring is a one-time cost. However, almost all location-based services involve dynamic data and thus tree updates. Following the STIG approach, the cost of handling updates will be extremely high as transmission of the entire tree is needed every time there is an update to the data. Another drawback of this approach is that query processing cannot proceed without CPU intervention.

In this paper, we present a GPU-based Parallel Indexing framework for Concurrent Spatial (G-PICS) query processing. G-PICS processes a group of spatial queries at a time, with each query assigned to a thread. Query processing is accomplished in two steps. In the first step, each query follows the traditional tree search path and registers itself to all leaf nodes intersecting with its query range. In other words, no data points are retrieved yet. In the second step, we follow a *query-passive* design: for each leaf node, we scan its data records only once, and distribute them to the list of queries pre-registered with that leaf node. The highly-organized access to data records yields great locality therefore can make good use of GPU cache. Meanwhile, all the accesses to the global memory are coalesced. G-PICS supports major spatial query types such as spatial point search, range search, within-distance search, k-nearest neighbors, and spatial joins. In addition, G-PICS encapsulates a highly efficient parallel algorithm for constructing spatial trees on GPUs. Furthermore, G-PICS provides an efficient parallel update procedure on GPUs to support dynamic datasets. We conduct comprehensive experiments to validate the effectiveness of G-PICS. Our experimental results show performance boosts up to 50X (in both throughput and query response time) over best-known parallel GPU and parallel CPU-based spatial query processing systems.

Paper Organization: In Section 2, we review the work related to the current problem; in Section 3 we introduce the new tree construction algorithm developed in G-PICS; in Section 4, we demonstrate the query processing technique in G-PICS; in Section 5, we elaborate our algorithm to support dynamic updates in G-PICS; in Section 6 we evaluate the performance of G-PICS, and in Section 7 we conclude the paper.

2 RELATED WORK

In the past decade, researchers have taken advantage of the computing capabilities of GPUs to speed up computational tasks in many application domains. In the database field, GPUs were exploited in relational operations such as aggregations [15], and join [18]. Significant speedups were reported as a result of integrating GPUs with databases in processing spatial operations [5]. Regarding the focus of this research, most of the previous work that parallelized queries on GPUs focused on parallelizing individual spatial queries instead of treating the entire workload as a whole. Most of the related work that used spatial data structures on GPUs focused on computer graphics applications [13, 21, 22, 42]. The indexes reported in such work are only for triangle partitioning and cannot be used for spatial partitioning. Some other work used R-tree data indexing for processing spatial range search queries on GPUs [39, 40]. Such work did not address key issues such as efficient construction of a parallel R-Tree on GPUs, and finding a solution to distribute the parallel tasks uniformly and efficiently among GPU threads. Since quadtree data structure was shown to be highly compatible with parallel architecture especially with GPUs [20, 28, 41], G-PICS uses a point-region (PR) quadtree as the partitioning data structure in 2-D space and oct-tree in 3-D space, although our ideas can also be applied to other spatial data structures.

On the other hand, spatial data structure construction and updating on GPUs are missing from most existing spatial query processing on GPUs. In [28] an algorithm for parallel construction of Bounding Volume Hierarchies (BVHs) on GPUs was developed. PR quadtrees are simplified variations of BVHs in 2-D space. The idea introduced in [28] was later used in [1, 24, 27] to convert the PR quadtree construction on GPUs to an equivalent level-by level bucket sort problem – quadtree nodes are considers as buckets and input data points are sorted into the buckets based on their locations at each level. In the suggested approaches, the nodes' partitioning process at each level is parallelized by applying an out-place quadtree bucket sort algorithm on the data list of the nodes that have to get partitioned in that level– nodes that number of points in them exceed maximum node capacity (*MC*) before reaching maximum height (*MH*) of the tree. In [27], the parallelization at each level is done by assigning one thread to each node that should get partitioned in that level; since the parallelization at higher level is poor, the first few levels usually are built on CPU. In order to improve this problem, in [24] each block of threads is assigned to one node that should get partitioned at higher levels of the tree. This idea was optimized by Nvidia to build a quadtree [1], in which each block of threads is assigned to a node that has to get partitioned, and each warp in a block works independently on building a child node in the next level. Then, each overflow child node launches another block of threads to build the next level of the tree from that node using CUDA dynamic parallelism. However, since the number of non-empty nodes is not known in advance, in [24, 27] in order to allocate node memory for the next level of the tree, four times of the number of nodes at the current level will be allocated for the next level. In [1], node memory allocation is done in advance by allocating the the maximum number of possible nodes based on the input *MH*. There is no previous work directly targeted at parallel data updates on spatial trees. In [31] a sequential algorithm on CPUs for updating PR quadtrees in which the maximum node capacity is equal to one was proposed. Their simulation mainly concentrates on consecutive phenomena such as fluids and smoke, in which if a point moves in a tree, it usually moves to its parent's sibling quadrants. The proposed updating procedure adaptively divides or merges the simulation area based on data movement. They show their approach outperforms two existing solutions for handling the updates – constructing the tree from scratch, and deleting a point from its current location and inserting to a new location.

3 QUADTREE CONSTRUCTION IN G-PICS

As discussed earlier, in STIG, the tree is constructed in host memory and then transferred to the GPU. The cost of building the tree on host memory is high. More critical issue is the overhead of transferring the tree from host to GPU - with its microsecond-level latency and 10GB/s-level bandwidth [3], the PCI-E bus is the weakest link in the entire GPU computing platform. Therefore, the first step towards enabling G-PICS for efficient spatial query processing lies in efficiently building a tree data structure within the GPU. In this paper, we focus on the PR quadtree for indexing 2-D data as an example while our approach can be extended to other types of quadtrees and oct-trees for 3D data. A PR quadtree is a type of trie for spatially indexing data points in 2-D space, in which each internal (link) node has at most four children. To construct a quadtree, two user-specified parameters should be given: *MC* and *MH*. Each node has a maximum capacity (*MC*), which is the maximum number of data points in a node. If data points in a node exceeds *MC*, the node is partitioned into four equal-sized child nodes. This decomposition continues until there is no more node to partition, or maximum height (*MH*) of the tree is reached [17, 34]. Nodes that got partitioned are link nodes, and others are leaf nodes. Oct-trees are analog of quadtrees in 3-D space.

There are unique challenges in parallel construction of a quadtree on GPUs. The traditional way for such is done by parallelizing the nodes' partitioning process level-by-level [24, 27]. Clearly, the scale of parallelism is very low at higher levels. Moreover, the total number of non-empty nodes in the tree is generally not known in advance. This issue causes a major problem as dynamic memory allocation on the thread level carries an extremely high overhead [18]. The easiest solution to tackle this problem which was adapted in the previous work is allocating four times of the number of nodes in the current level before developing the next level of the tree. Consequently, empty nodes will expand until the very last level of the tree. This results in inefficient use of (global) memory, which is of limited volume on GPUs, and becomes more crucial when dealing with skewed datasets. In addition, the main design goal of G-PICS is to allow efficient query processing; hence, placing data

Algorithm 1: G-PICS Tree Construction Routine

Var: $splitNum$ (number of nodes to be partitioned) $\leftarrow 1$,
$Split$ (array to track nodes' split status),
C_{node} (array shows current node for each data point) $\leftarrow 0$,
Cur_{level} (current level developing in the tree) $\leftarrow 1$

1: $Split[0] \leftarrow$ True
2: **while** $Cur_{level} < MH$ and $splitNum > 0$ **do**
3: Cur_{level}++;
4: Tree-Partitioning on GPU;
5: update $splitNum$
6: **end while**
7: Node-Creation on GPU;
8: Point-Insertion on GPU;

Algorithm 2: Tree-Partitioning on GPU

Global Var: $Input$ (array of input data points),
$Counter$ (array to reflect the number of points in each node)
Local Var: t (Thread id),
$L_{counter}$ (counter value after adding a point to a node)

1: **for each** $Input[t]$ **in parallel do**
2: **if** $Split[C_{node}[Input[t]]]$ == True **then**
3: $C_{node}[Input[t]] \leftarrow$ find position of $Input[t]$ in the children of $C_{node}[Input[t]]$
4: $L_{counter} \leftarrow$ atomicAdd($Counter[C_{node}[Input[t]]]$, 1)
5: **if** $L_{counter}$ == MC+1 **then**
6: $Split[C_{node}[Input[t]]] \leftarrow$ True
7: **end if**
8: **end if**
9: **end for**

points belonging to a leaf node in consecutive memory locations is the prerequisite of achieving coalesced memory access (Section 4).

Overview of G-PICS approach: To address above challenges, we propose a top-down parallel tree construction on GPUs that achieves a high level of parallelism at all times. Furthermore, our approach resolves the problem of empty node expansions, and guarantees coalesced memory access by storing the data points belonging to a leaf node in consecutive memory locations.

G-PICS tree construction solves the empty node expansion problem on GPUs by delaying the actual node memory allocation until the exact number of non-empty nodes in the tree is determined. In particular, in the beginning, it is assumed that the tree is a full tree according to its MH – in a full quadtree all the intermediate nodes in the tree have exactly four children. The maximum number of nodes in a full quadtree with height of H, can be calculated as follows: $\sum_{i=0}^{i=(H-1)} 4^i = (4^H - 1)/(4-1) = (4^H - 1)/3$. Each node in a full quadtree has an ID, which is assigned based on its position in the tree. Starting from the root node with the ID equals to zero, and allocating the directions based on the children location (ranging from 0 to 3), an ID for each node is determined as follows: $Node_{id} = (Parent_{id} * 4) + direction + 1$, in which $Parent_{id}$ is the ID of the node's parent. Figure 3 shows a small (i.e., $MC = MH = 3$) example of G-PICS quadtree construction on a set of fourteen data points distributed in a region.

The main idea behind G-PICS is a new parallelization paradigm for quadtree construction on GPUs. This paradigm maintains a high level of parallelism by novel workload assignment to GPU threads. Instead of binding a thread to a tree node, each G-PICS thread is assigned to one input data point, and the process of locating the node to which each point belongs is parallelized. Each thread keeps the ID of such nodes and such IDs are updated at each level till a leaf node is reached. The tree construction (Algorithm 1) is done in three steps via three GPU kernels: Tree-Partitioning, Node-Creation, and Point-Insertion. In the following discussions, we assume the entire tree plus all the data can fit into the global memory.

Tree Construction Routines: The Tree-Partitioning kernel (Algorithm 2) is launched with N threads, with each thread working on one data point. Starting from the root node, each thread finds the child node to which its assigned point belongs, and saves the child node ID. Meanwhile, the thread also increments the value of the current node's data point counter. Since such counts (i.e., counter array) are shared among all threads, we use *atomic instructions* provided by CUDA programming environment to update the counts and maintain correctness. When the counts are updated, if a node's data count exceeds MC and MH of the tree has not been reached yet, the corresponding value in the split array will be set, meaning that the node should get partitioned in the next level of the tree. Upon finishing operations at one level (for all threads), the following information can be seen from auxiliary arrays: current node array indicates the nodes to which data points belong, node counter array reflects the number of data points in each node, and split array indicates if each node has to be partitioned. If there are nodes to be partitioned, the same kernel is executed again to develop the next level of the tree. For example, in Figure 3, there are two nodes – N_2 (with 5 points) and N_3 (with 7 points) – to be partitioned when level 2 of the tree is built. The kernel is relaunched with three new auxiliary arrays, the length of which correspond to the number of the child nodes of only N_2 and N_3. Likewise, data counts and the split of the nodes in this level are updated. This routine will continue until there are no more nodes to be split, or the MH of the tree is reached. This tree construction paradigm maintains a high level of parallelism by having N active threads at each level.

The Node-Creation kernel (Algorithm 3) is called to create the actual non-empty nodes in the tree. Having finished the previous step, the following information is known: each point has the leaf node to which it belongs, the total number of non-empty nodes in the entire tree with their types (leaf or link), and the total number of points in each leaf node. Therefore, the required information for creating the nodes in a parallel way and without wasted space is known. Consequently, the Node-Creation kernel is launched with as many threads as the number of nodes, and non-empty nodes are constructed in parallel. While building nodes, point memory for leaf nodes data list in consecutive memory locations is also allocated. In Figure 3, the total number of non-empty nodes is 11 (versus the full quadtree which has 21 node in this example). Consequently, space for only 11 nodes is allocated in this example. Out of the 11 nodes, eight are leaf nodes (split is False), and three of them are link node (split equals to True).

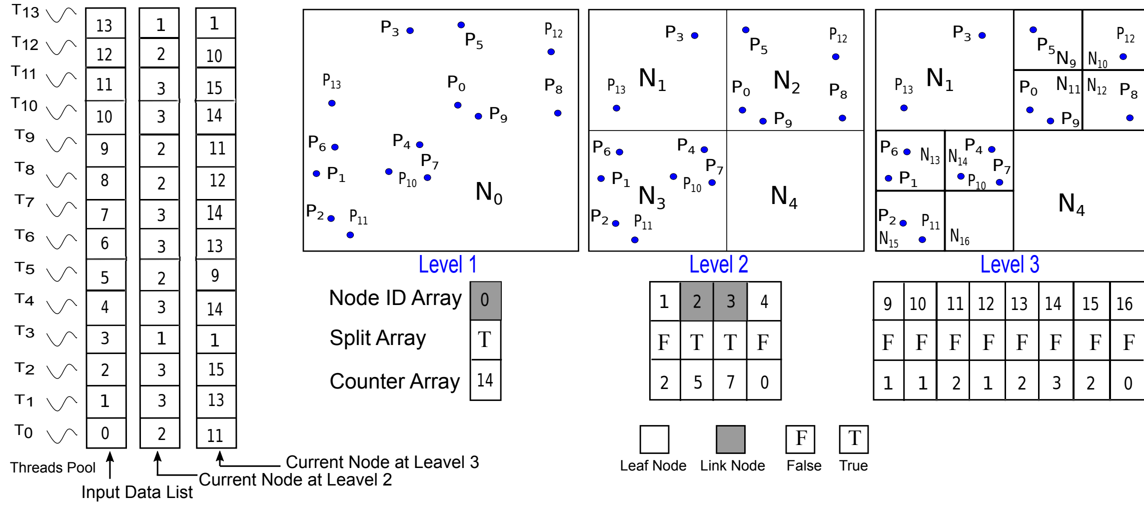

Figure 3: An example of quadtree construction in G-PICS with $MC = MH = 3$. Auxiliary arrays with the length equals to the maximum number of nodes in a full tree are allocated on GPU and deleted when the tree construction is completed.

Algorithm 3: Node-Creation on GPU

Global Var: $Node_{ID}$ (array holding IDs of nodes), $leaf_{datalist}$ (array to store data points in leaf nodes)
Local Var: t (Thread id)
1: **for each** non-empty $Node_{ID}[t]$ **in parallel do**
2: create node $Node_{ID}[t]$
3: **if** $Split[Node_{ID}[t]]$== False and $Counter[Node_{ID}[t]] > 0$ **then**
4: Allocate consecutive memory of size $Counter[Node_{ID}[t]]$ in $leaf_{datalist}$
5: **end if**
6: **end for**

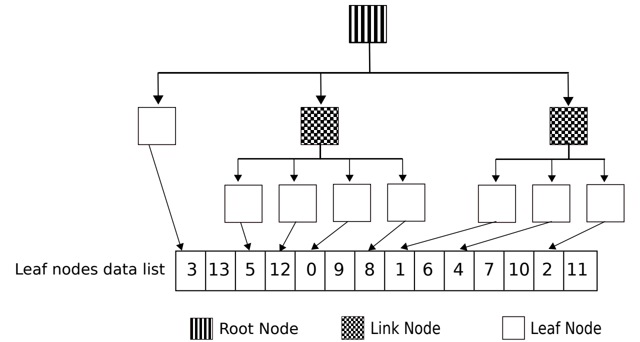

Figure 4: Final quadtree built based on the data inputs in Figure 3

The Point-Insertion kernel (Algorithm 4) is called to insert the input data points to tree. Each thread takes one point, and inserts that point to its corresponding leaf node's data list. Having this setup, all the points in each leaf node are saved in consecutive memory locations. The input data points in a quadtree have two dimensions (x, and y). To ensure coalesced memory access in query processing, the data lists should be saved using two arrays of single-dimension values rather than using an array of structures which holds two-dimensional data points. The final quadtree structure built using the aforementioned algorithm is shown in Figure 4, in which each leaf node points to the beginning of its data list in the leaf nodes data list array.

Algorithm 4: Point-Insertion on GPU

Local Var: t (Thread id)
1: **for each** $Input[t]$ **in parallel do**
2: insert $Input[t]$ to $leaf_{datalist}[C_{node}[Input[t]]]$
3: **end for**

4 G-PICS QUERY PROCESSING

In this section, we elaborate on query processing algorithms in G-PICS. As mentioned earlier, G-PICS supports multiple spatial queries running concurrently to improve resource utilization and overall efficiency. G-PICS provides support of the following major spatial query types: (1) *spatial point search*, which retrieves the existence of an object in the input data set, (2) *range search*, which finds a set of data objects within specific region of rectangular shape, (3) *within-distance search*, which retrieves objects within a certain circle in which the input search query is the center of that circle and search distance determines the radius of the circle, (4) *k-nearest neighbors*, which is a direct generalization of the nearest neighbor problem, where k closest objects to the input query will be retrieved, and (5) *spatial join*, which retrieves all pairs of objects in the input dataset satisfying the search condition. A typical spatial query is processed in two steps:

(1) leaf nodes satisfying the search conditions are identified; and
(2) all data points in the identified nodes are examined to determine the final results.

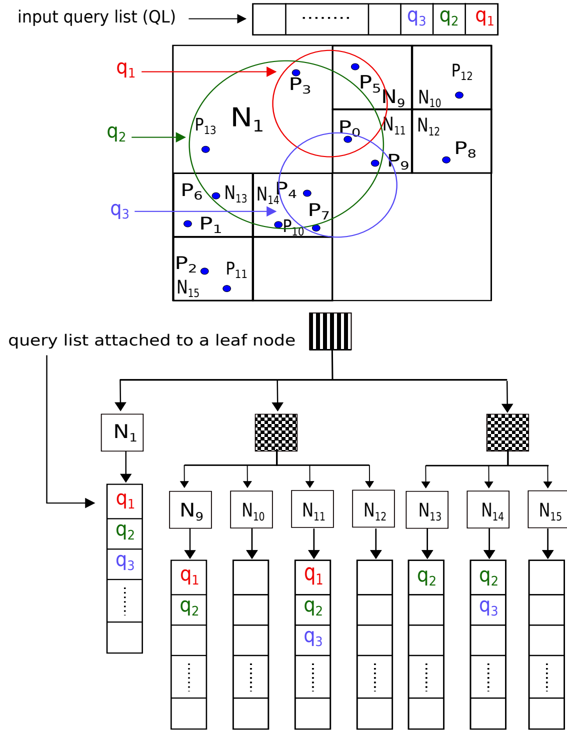

Figure 5: An example of registering three within-distance search queries into the query lists of leaf nodes

Recall that in the first step, using the traditional tree search is necessary to achieve logarithmic work efficiency. Moreover, in the second step, reading the data records should be done through global memory, and the same data record can be accessed many times by different queries in a workload. To tackle these issues, G-PICS introduces a new paradigm for processing multiple spatial queries concurrently on GPUs in two steps. All the spatial input queries in a workload are considered as input query list (QL). All the input queries in the QL share the same processing strategy using the two-step query processing. In the first step, all queries have to find their intersecting leaf nodes using a traditional logarithmic tree search, and in the second step, they have to process their intersecting leaf nodes' data lists to output their final results. This common strategy provides the opportunity for batch query processing. The most expensive part in query processing is reading the intersecting leaf nodes' data lists for each query from GPU's global memory. To minimize this cost as much as possible, we use a common scanning process to read the data lists. Therefore, queries that access the same data list can be served together. To this end, we design a new query processing strategy in G-PICS in which in the first step of query processing, queries intersecting the same leaf node are grouped together. Then, in the second step, queries in each group are processed together to minimize accesses to global memory, and take advantage of the other available low-latency memories on GPUs.

In order to group the queries together, a set of query lists are added to the tree data structure. Each query list is attached to a leaf node for saving the list of queries intersecting that leaf node for processing. Two GPU kernels are designed in G-PICS to perform the two-step query processing: Leaf-Finding, and Leaf-List-Processing.

Algorithm 5: Leaf-List-Processing on GPU

Local Var: b (Block id), t (Thread id in a block), M (total number of points in the leaf[b]), lqL (query list attached to the leaf[b]), sL (list of points in shared memory)

1: $sL \leftarrow$ load $leaf_{datalist}$[leaf[b]] from global memory to shared memory in parallel
2: **for each** $lqL[t]$ **in parallel do**
3:   **for** $i = 1$ to M **do**
4:     $d \leftarrow$ computeSearchFunction($lqL[t]$, $sL[i]$)
5:     **if** d meets the search condition **then**
6:       Add $sL[i]$ to the Output list of $lqL[t]$
7:     **end if**
8:   **end for**
9: **end for**

Step I - Query Registering: In the Leaf-Finding kernel, each thread takes one query from QL, and registers that query to the query lists of all the identified leaf nodes through traditional tree search algorithm. After finding the intersecting leaf nodes for all queries, each leaf node has a query list of registered queries that should retrieve the data records in that leaf node to output their final results. Figure 5 shows an example of registering queries into the query lists of the leaf nodes for three within-distance search input queries.

Step II - Leaf Node Processing: To process the registered queries in the query list of the leaf nodes, the Leaf-List-Processing kernel is launched with as many GPU blocks as the number of leaf nodes. Each leaf node is assigned to one GPU block. The maximum number of active threads in each block is equal to the number of registered queries in the query list of the leaf node that is assigned to that block. In order to output the results, all the queries in a leaf query list have to read the data records in that leaf node and based on their query types perform the required computation. Therefore, in each GPU block, first, all the data points belonging to the leaf node assigned to that GPU block is copied from global memory to shared memory in parallel. Shared memory is much faster than global memory - its access latency is about 28 clock cycles (versus global memory's 350 cycles) [2]. The copying from Global memory to the shared memory is not only parallelized, but also coalesced because points in each leaf node are saved in consecutive memory locations. Using this strategy, the number of accesses to each leaf node data lists in global memory are reduced to one. This is in sharp contrast to the traditional approach that retrieves each leaf node once for each relevant query. Having copied all the points in each leaf node data list to the shared memory, each active thread in that block takes one query from the query list attached to its corresponding leaf, copies the query point into a register, calculates the computation function (which is usually Euclidean distances computation) between that query point and all the points in that leaf node (which are all located in share memory), and outputs those that satisfy the search criteria. This step is shown in Algorithm 5.

Algorithm 6: Second step of spatial join

Local Var: b (Block id), t (Thread id in a block), M (total number of points in the leaf[b]), lqL (query list attached to the leaf[b]), N (total number of registered leaf node in lqL), d (search distance), dL_i (list of points in leaf node i)

```
1:  dL_b ← leaf_datalist[leaf[b]]
2:  for each dL_b[t] in parallel do
3:      for i = t+1 to M do
4:          r ← computeDistanceFunction(dL_b[t], dL_b[i])
5:          if r<= d then
6:              Add dL_b[i] to the Output list of t
7:          end if
8:      end for
9:  end for
10: for j = 1 to N do
11:     dL_j ← leaf_datalist[leaf[lqL[j]]]
12:     for k = 1 to dL_j.numberOfPoints do
13:         r ← computeDistanceFunction(dL_b[t], dL_j[k])
14:         if r<= d then
15:             Add dL_j[k] to the Output list of t
16:         end if
17:     end for
18: end for
```

Query Implementation Details. The point search, range search, and within-distance search are implemented following the 2-step approach in a straightforward way. The kNN in G-PICS is treated as a within-distance search followed by a k-closest selection from the within-distance search result set. The within-distance search is initialized with a small radius. If the number of output items for a query is less than k, the within-distance search will be performed again with a larger radius until it has enough (k) outputs. We implement a special type of spatial join named the 2-Body constraints problem which retrieves all pairs of objects that are closer than a user-specified distance (d) from each other. In the first step of the spatial join search, each leaf node registers itself to the query list of all other leaf nodes with distance less than or equal to d. Then, in the second step of the search (Algorithm 6) all matching pairs are retrieved and outputted.

Outputting Results. A special challenge in GPU computing is that, in many applications, the output size is unknown when the GPU kernel is launched. Examples of such in G-PICS are the number of output results in a typical window range query, within-distance query, and spatial joins. The problem comes from the fact that memory allocation with static size is preferred - in-thread dynamic memory allocation is possible but carries a huge performance penalty [18]. This causes a serious bottleneck in those applications. A typical solution is to run the same kernel two times; in the first round, output size is determined. Then, in the second run, the same algorithm will be run again and output will be written to the memory allocated according to the size found in the first round. In G-PICS, we utilized an efficient solution introduced in our previous work [33], which allows our algorithms to compute and output the results in the same round for those categories of queries that their output size is unknown in advance using a buffer management mechanism. This design minimizes threads synchronization. In this design, an output buffer pool with a determined size is allocated. The allocated memory is divided into small chunks called **pages**. In order to have access to the location of the first available page in the buffer pool, a global pointer (GP) is kept. Each thread gets one page from the buffer pool and outputs its results to that page. It also keeps track of its own local pointer to the next empty slot within that page. Once a thread has filled a page completely and has more results, it will get a new page from the buffer pool by increasing a GP using the GPU atomic add operation. Using this solution, conflicts among threads is minimized because GP is updated only when a page is completely filled.

5 TREE UPDATES IN G-PICS

While support for dynamic data inputs are missing from existing GPU-based spatial tree work, G-PICS provides an efficient parallel update procedure on GPUs to support dynamic datasets. Moving a data point in a quadtree can be summarized by a deletion followed by an insertion. However, this operation can be expensive considering the nature of quadtree structure and data movements, especially when there are group movements. By moving a data point in the tree, the tree data structure, and leaf nodes' data lists should be updated accordingly. Both of these operations are costly because dynamic memory allocation on the thread level carries an extremely high overhead on GPUs. At the end of each move, a data point can either stay in the same leaf node or move into another one. After all movements, the number of points in some leaf nodes may exceed *MC*. Consequently, if *MH* is not reached, those nodes should be partitioned, and points in them moved to their children. Alternatively, sibling leaf nodes could lose data points and their total number of data points go below *MC*. In these cases, the siblings should be merged together, and data points in them moved to their parent nodes. Moreover, there may be cases that points move into an empty node. It was mentioned earlier that empty nodes are not materialized in our tree structure. Therefore, first new nodes should be built to represent the region that was empty in that direction, and then points should be inserted. All these changes usually happen at the lowest levels of the tree.

Figure 6 shows an example of these scenarios: Figure 6(a) shows the tree structure before data movement and Figure 6(b) shows that after movements. The movements in this tree are as follows: P_5 and P_0 moved to N_1, P_9 moved to an empty node, and all other points did not move out from their last-known leaf node in the tree. After all these movements, the number of points in N_1 exceeds *MC*. Therefore, N_1 should get partitioned, and points in that node moved to its children. On the other hand, the total number of points in N_9, N_{10}, N_{11}, and N_{12} is less than *MC*. Therefore, these four sibling nodes should be merged together, and points in them moved to their parent (N_2). Since P_9 moved to an empty node, first a node should be built in that region. The final tree structure after all the updates is shown in Figure 6(b).

Bulk Updates in G-PICS. We design a bottom-up parallel algorithm on GPUs to bulk update the tree structure. This, again, reflects our strategy of concurrent processing of queries except now a query is an update. At first, the new position of all the input data points are checked in parallel to see if they moved out from their last-known leaf node in the tree. If there is at least one movement, the tree structure should be updated accordingly. Updating the tree

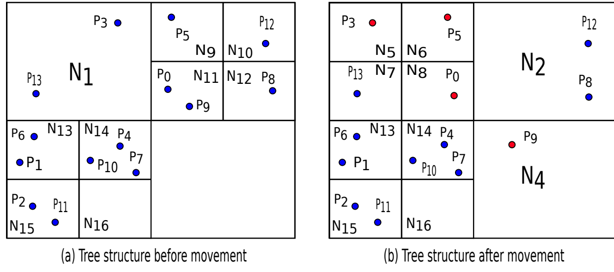

Figure 6: An example of quadtree update for $MC = 3$ and $MH = 4$

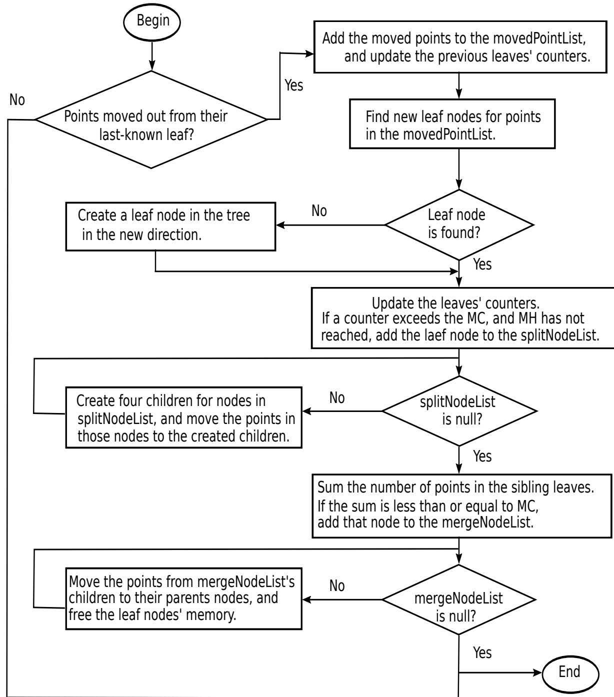

Figure 7: Quadtree update procedure in G-PICS

structure may lead to partitioning, merging, or creating nodes in the tree. Several GPU kernels are designed to update the tree in parallel on GPU as follows: Movement-Check, Leaf-Finding, Node-Partitioning, Node-Merging, and Leaf-Lists-Update.

The Movement-Check kernel checks if each data point has moved out from its last-known leaf node. In this kernel as many threads as N are launched, each thread takes one point, and checks if that point has moved out from its last-known leaf node. In case of movement, the corresponding thread adds the point to the list of moved data points, and updates the counter of the number of points in the last-known leaf node. Having finished this kernel, if there is at least one movement, the tree structure and the data lists of leaf nodes have to be updated accordingly.

The Leaf-Finding kernel is called if the list of moved data points is not empty. This kernel is launched with as many threads as the number of moved data points. Each thread gets one point from the list of moved data points, finds a new leaf node for that point in the tree, updates the number of points in the new leaf node, and saves that leaf node as a new location for its corresponding point. If by updating the number of points in a leaf node the total number of points in that leaf node exceeds MC and MH has not been reached, that node is added to the list of nodes that should be partitioned. On the other hand, there may be points that move to empty nodes. These points are added to another list; then, first new nodes are created in the corresponding directions, and afterwards those points are added to the newly-created nodes. When data points spread out evenly in the input space, this case is very rare; however, to have a robust design, this case is also considered in G-PICS. New nodes creation is done in parallel, and data points insertion into those nodes is also done in parallel. Meanwhile, if by updating the nodes' counters, the number of points in a node exceeds MC, that node is added to the list of nodes that should be partitioned.

The Node-Partitioning kernel is called if the node partition list is not empty. This kernel is launched with as many threads as the number of nodes in that list. Each thread takes one node, creates child nodes for that node, and sets up the parameters for them. The points belonging to the partitioned nodes should be moved to the data list of the newly-built child nodes. There are two groups of points that may belong to nodes in the partition list: points that previously belonged to those leaf nodes, and did not move out, and points from moved data list that moved to those nodes during their movement. To minimize the computation and maximize the efficiency, extra operations should be minimized as much as possible. Therefore, just these two groups of points are considered for the sake of finding new locations in the children of partitioned nodes. While updating the counters of child nodes, if a node becomes qualified for partitioning, that node is added to the split nodes' list. The above steps will repeat for new nodes in the split list until there is no more nodes to be partitioned.

On the other hand, while some nodes are to be partitioned, there may be other nodes that have to be merged. Except the leaf nodes that get partitioned, other leaf nodes have the potential of getting merged. The Node-Merging kernel considers those leaf nodes by checking the sum of the counter of sibling leaves. If the total number of points in those nodes become less than or equal to MC, they are added to the list of nodes that has to be merged. Siblings in this list are merged together, and their data points are moved to their parent. Consequently, those parent nodes become the new leaves. The algorithm of moving points in these nodes is similar to the one in Node-Partitioning kernel. However, the only difference is that in this part the points should be moved to the parent of their last-known leaf node.

Having finished all these steps, each point has the final leaf node to which it belongs, and each leaf node has the total number of points in its data point list. Since deletion and insertion by shifting is very costly on GPUs, data lists in leaf nodes can be updated by reinsertion from scratch. Therefore, Leaf-Lists-Update kernel is called to insert data points into their corresponding leaf nodes using the same procedure mentioned in the tree construction. The work-flow of the entire tree update procedure is shown in Figure 7.

6 EXPERIMENTAL RESULTS

In this section, we present empirical evaluation of the performance of G-PICS. For that, we implemented a CUDA version of G-PICS as well as processing algorithms for the following queries: window-based range search, within-distance search, k-nearest neighbors, spatial point search, and spatial joins. We conduct experiments on a workstation running Linux (Ubuntu 16.04 LTS) with an Intel Core i9-7920X 2.90GHz CPU with 12 cores, 64GB of DDR4 3200 MHz memory, and an Nvidia Tesla P100 GPU (16GB global memory).

As mentioned earlier, G-PICS is designed to process a (large) group of spatial queries at a time. Two baseline algorithms for processing multiple concurrent queries are developed: a parallel CPU algorithm (P-CPU), and M-STIG. M-STIG is a task parallel GPU algorithm for processing multiple queries at a time developed following the descriptions in STIG [12] for processing one query at a time. We implement the P-CPU algorithms in the C programming language using OpenMP. Note that P-CPU is highly optimized, and performs a traditional tree search in the first step of query processing to bear the logarithmic work efficiency. Moreover, additional techniques for improving the P-CPU performance using OpenMP are also applied including: choosing the best thread affinity for the thread scheduler, best thread scheduling mode, and best number of active threads per simulation. Furthermore, both GPU implementations (G-PICS and M-STIG) are highly optimized in terms of efficient use of registers, choosing the best block size in each GPU kernel. Such optimizations are done according to our previous work in GPU kernel modeling and optimization [29]. G-PICS tree construction performance is evaluated by comparing the performance with parallel quadtree construction code developed by Nvidia which is available at [1]. All the experiments are conducted over two types of data: a real dataset [10] generated from a large-scale molecular simulation study of a lipid bi-layer system, [1] and synthetic datasets. A visualization of the real dataset indicates that the data points are almost **uniformly distributed** in space. The synthetic data, on the other hand, is generated following a Zipf distribution under different orders to generate **highly skewed** datasets.

6.1 Tree Construction in G-PICS

The parallel tree construction codes introduced by [24, 27] are not publicly available. Therefore, we implemented the algorithms following their descriptions; however, their algorithms showed very poor performance comparing to ours (G-PICS achieves more than one thousand performance speedup over them). This indicates that, due to very poor level of parallelism and inefficient memory swapping for allocating points into relevant quadtree nodes at each level the above academic codes can hardly represent the state-of-the-art in GPU quadtree construction. In such experiments, only the tree construction time is measured and compared. Consequently, to have a more meaningful evaluation, we compare G-PICS tree construction with the parallel tree construction developed by Nvidia [1]. Figure 8 shows the G-PICS tree construction time, and performance speedup of G-PICS tree construction over Nvidia [1]. As shown, G-PICS clearly outperforms the Nvidia (up to 53X) in all cases. By

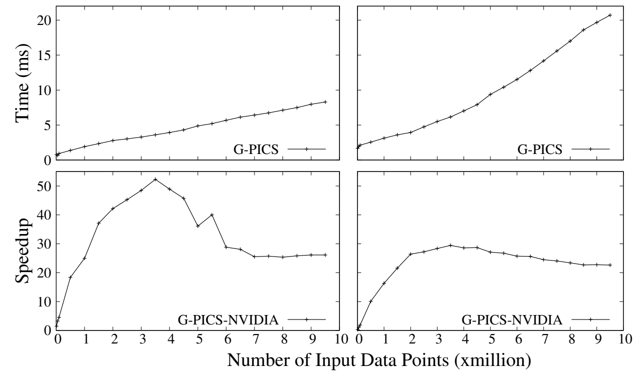

Figure 8: G-PICS tree construction time, and speedup of G-PICS tree construction over the Nvidia code. Left: real dataset; Right: skewed dataset

increasing the number of input data points, G-PICS speedup increases, and it remains constant at very large input datasets. Note that in G-PICS points belonging to a leaf node are saved in consecutive memory locations. For the real data with a nearly uniform spatial distribution, G-PICS shows better performance advantage. For the skewed data, G-PICS Point-Insertion kernel takes longer to run. The main reason for G-PICS' high performance is the new parallelism paradigm that maintains a high level of parallelism at all times. Moreover, G-PICS does not materialize the empty nodes. In addition, while building the tree in G-PICS, if *MH* is reached, leaf nodes at *MH* will accommodate more points than *MC*. However, in Nvidia the algorithm will stop working. Therefore, the Nvidia tree construction should be initialized with a large *MH* to make sure the tree construction works properly which leads to more empty node expansion.

6.2 Spatial Query Processing in G-PICS

As the query processing in G-PICS and M-STIG is done in two steps, the performances of both steps, as well as the overall performance are compared. Each dataset has 9.5 million data points, which are indexed using the parallel quadtree construction mechanism of G-PICS. The *MC* is set to 1024, and *MH* is set to 12. The batch query processing performance is evaluated for different number of input queries. In the following text, we report and analyze two performance metrics: (1) the relative total processing time (i.e., speedup) used for processing all such queries by G-PICS over baseline codes (i.e., M-STIG and P-CPU); and (2) the minimum, maximum, and average response time of all queries achieved by all three programs.

6.2.1 Range Search. Range (window-based) search queries are highly compatible with quadtree data structure, since in the first step of the search, the intersection of rectangles (input query and quadtree nodes) should be evaluated. The output size of a typical range search is unknown in advance; we therefore use the buffer pages solution discussed in Section 4 to do the computation and outputting the results in the same round (in both G-PICS, and M-STIG). Figure 9 shows G-PICS query processing time along with its speedup over the baseline codes in processing range search queries.

[1]The dataset contains the coordinates of 268,000 atoms recorded over 100,000 time instances. We superimpose the coordinates of different time instances to generate data of an arbitrary (large) size for experimental purposes.

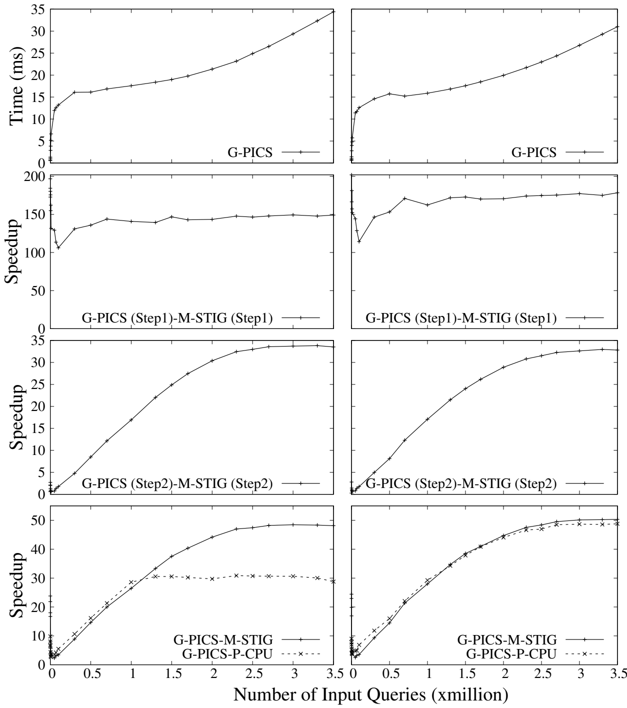

Figure 9: G-PICS range search query processing time, and speedup of G-PICS over M-STIG and P-CPU in processing range search queries. Left: real dataset; Right: skewed dataset

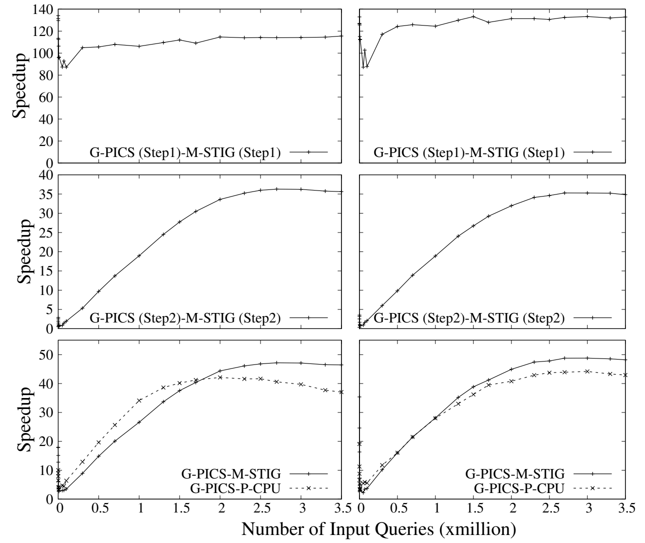

Figure 10: Speedup of G-PICS over M-STIG and P-CPU in processing within-distance search queries. Left: real dataset; Right: skewed dataset

However, in other spatial query processing due to page limits just speedup is shown. Figure 9 shows that logarithmic tree search in the first step noticeably outperforms the brute-force leaf search under any circumstances. The performance speedup is even more remarkable in skewed dataset because the input queries may intersect with fewer leaf nodes. The considerable performance boost is certainly in conformity with our understanding of using logarithmic tree search over brute-force. The performance of the second step of G-PICS (Figure 9) is more obvious as the number of concurrent queries increases. It starts with a small number under very low concurrency, climbs rapidly (in a linear manner) by increasing the number of input queries in both datasets, and stabilized gradually. This rapid growth is the result of sharing shared memory by more queries and reducing the total number of accesses to global memory. Finally, the overall speedup of G-PICS over M-STIG and P-CPU (Figure 9) increases substantially by increasing the number of input queries and remains stable eventually. This level off is the result of having more queries in each query list. Consequently, it takes longer to process the queries and output the results.

6.2.2 Within-Distance Search and kNN. It was mentioned earlier that kNN in G-PICS is treated as a within-distance search, followed by performing the k-closest selection in the results of within-distance search. The selection kernel for finding kNN is the same in G-PICS and M-STIG. Therefore, within-distance search operation is a determining factor in the performance evaluation. Similar to range search, the output size of a within-distance search is not known in advance; consequently the buffer pages solution is used for outputting the results. The range of intersect in a typical within-distance search is a circle versus a rectangle in a range search query. Therefore, the graphs in Figure 10 for evaluating the performance of within-distance search queries in G-PICS follow the same trend as those presented in range search.

6.2.3 Spatial Point Search. Comparing with other types of queries, there is less computation involved in processing point search queries. In the first step of the search, each query just intersects with one leaf node, therefore, logarithmic tree search speedup over brute-force leaf search presents the most significant improvement comparing to the other query types (Figure 11). On the other hand, although the speedup of the second step follows an upward trend, it shows less performance increase comparing with the second step of other query types. This reduction is because less computation involved in processing point search queries. However, the first step of M-STIG is very slow, therefore the overall performance speedup of G-PICS over M-STIG climbs sharply. Likewise, G-PICS shows a gradual upward performance rise over P-CPU. However, in processing this type of queries, P-CPU outperforms M-STIG. Note that comparing to the other query types, it takes much longer to see the level off in G-PICS point search speedup. This happens because in the first step of search each query is registered to one query list. Consequently, it takes longer to have a longer query list in each leaf node.

6.2.4 Spatial Join. In order to evaluate the performance of spatial join, we implemented the 2-Body constraints problem that retrieves all pairs of objects that are closer than a determined distance (d) from each other in the input domain. In processing this type of queries, the second step of the search for G-PICS and M-STIG is the same (Algorithm 6). Consequently, the first step of the

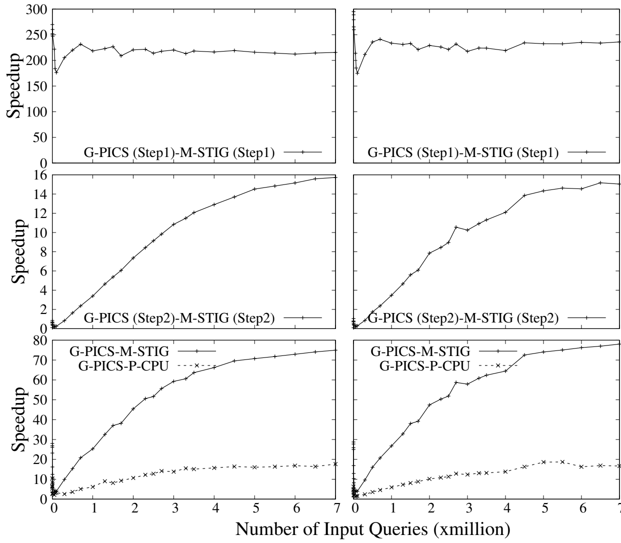

Figure 11: Speedup of G-PICS over M-STIG and P-CPU in processing point search queries. Left: real dataset; Right: skewed dataset

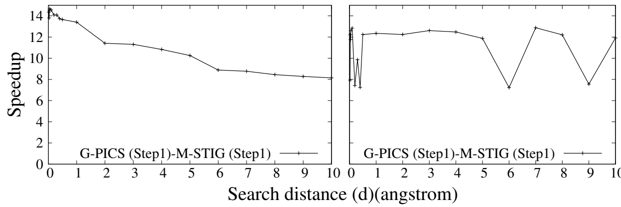

Figure 12: G-PICS spatial join performance speedup over M-STIG under d = 100 angstrom. Left: real dataset; Right: skewed dataset

search determines the overall performance. The cost of query registering in the first step is constant in M-STIG regardless of d. Figure 12 shows the overall performance comparison of the G-PICS over M-STIG with variable distances. In real dataset, achieved speedup in G-PICS decreases gradually by increasing d. This is the result of visiting more leaf nodes in the search. However, in the skewed dataset, this trend has more fluctuation because some parts of the tree has more data congestion due to data skewness.

6.2.5 Performance under low concurrency. As a special note, even when the number of input queries is small, G-PICS still outperforms M-STIG and P-CPU in processing all query types. In particular, under concurrency of one in processing range search and within-distance search, G-PICS shows a speedup of at least 20X and 10X over M-STIG and P-CPU, respectively. For point search query, the speedup is 28X and 7X over M-STIG and P-CPU, respectively. This demonstrates the importance of conducting logarithmic tree search.

6.2.6 Response Time Analysis. As discussed earlier, STIG processes one query at a time while G-PICS parallelizes many queries together. One might argue that while G-PICS achieves much higher throughput, it could suffer from poor (individual) query latency –

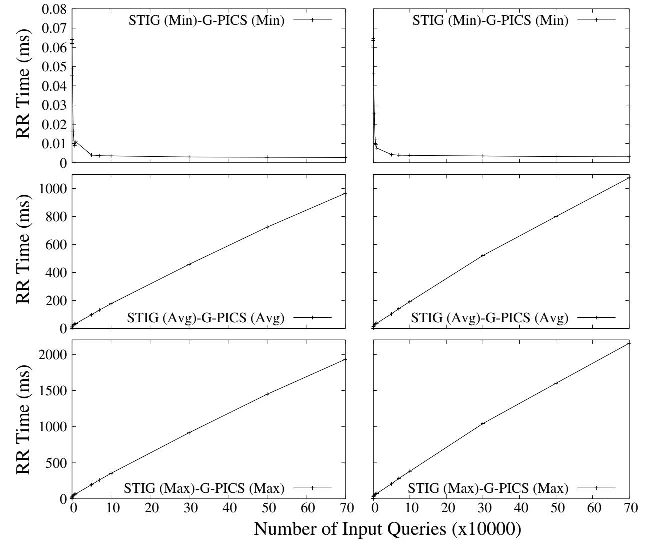

Figure 13: Relative Response (RR) Time of range search queries processed by G-PICS over that by STIG. Left: real dataset; Right: skewed dataset

queries have to wait till the batch is formed. This is true if we look at the minimum response time (Figure 13, top) among all queries processed: such number could be two orders of magnitude higher in G-PICS. Note that minimum response time among all queries is similar to average and maximum response time in G-PICS (due to batch processing) while the first few queries in STIG could have a very short response time. However, the average response time (Figure 13, middle) of G-PICS queries is much better than in STIG, with an improvement of nearly 1000X depending on the query numbers in QL. The maximum response time has the same trend as average response time (Figure 13, bottom), however, with much sharper rise. Therefore, although the first few queries in STIG run faster, it takes so long to process the entire workload. Therefore, in systems with concurrent queries G-PICS always outperforms STIG. The above data is collected under low concurrency and for range search queries only. However, in busier systems the improvement of response time by G-PICS is much more considerable. The same trends can be observed in processing other query types.

6.3 Tree Update Performance in G-PICS

In order to support dynamic datasets, the constructed tree structure should be updated based on the input data points movement. In order to evaluate the performance of the designed update procedure, each time we change the positions of a specific percentage of the input data points to new randomly-generated positions. Then, the tree is updated accordingly. To evaluate the performance, the time takes to update the tree is compared with time to construct the tree from scratch in G-PICS (as no other work can be found in GPU-based tree updates). Figure 14 shows the speedup of the tree update algorithm over tree construction from scratch. At low movement of input data, the G-PICS update procedure can be up to 150% faster than tree construction from scratch. By increasing the number

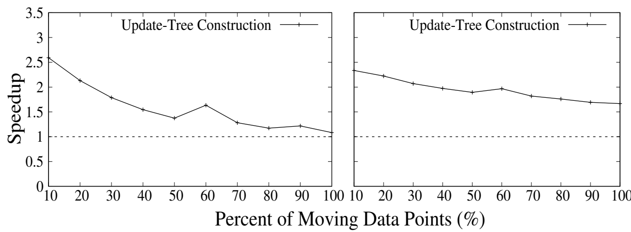

Figure 14: G-PICS update performance over G-PICS tree construction from scratch. Left: real dataset; Right: skewed dataset

of movements, the update performance decreases. However, even when all the input data points change their positions, the G-PICS update procedure is still more efficient.

7 CONCLUSIONS

In this paper, we argue for the importance of GPUs in spatial query processing, especially in applications dealing with concurrent queries over large input datasets. It was shown that parallelism and spatial indexing are the prerequisite of achieving high performance in processing such queries. To this end, we present a GPU-based Parallel Indexing framework for Concurrent Spatial (G-PICS) query processing. G-PICS provides a new tree construction algorithm on GPUs, which achieves a high level of parallelism and shows a magnitude performance boost over the best-known parallel GPU-based algorithms. Moreover, G-PICS introduces the new batch query processing framework on GPUs to tackle the low work efficiency and low resource utilization existing in current one-query-at-a-time approaches. G-PICS supports the processing of major spatial query processing, and shows a great performance speedup over the best-known parallel GPU and parallel CPU-based spatial processing systems. In addition, G-PICS provides an efficient parallel update procedure on GPUs to support dynamic datasets.

REFERENCES

[1] *Quad Tree Construction.* https://github.com/huoyao/cudasdk/tree/master/6_Advanced/cdpQuadtree

[2] 2014. *Analyzing GPGPU Pipeline Latency.* http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf

[3] 2017. *Low Latency PCIe Solutions for FPGA.* Technical Report. Algorithm in Logic.

[4] E. Alba, G. Luque, and S. Nesmachnow. 2013. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research* 20(1) (2013), 1–48.

[5] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. 2004. Hardware acceleration in commercial databases: A case study of spatial operations. In *VLDB-Volume 30.* 1021–1032.

[6] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. 2009. Experiences on processing spatial data with mapreduce. In *International Conference on Scientific and Statistical Database Management.* 302–319.

[7] S. Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth SIGMOD-SIGART.* 34–43.

[8] H. Chavan and M. F. Mokbel. 2017. Scout: A GPU-Aware System for Interactive Spatio-temporal Data Visualization. In *Proceedings of the ACM.* 1691–1694.

[9] H. Chen, J. Martinez, A. Kirchhoff, T. D. Ng, and B. R. Schatz. 1998. Alleviating search uncertainty through concept associations: Automatic indexing, co-occurrence analysis, and parallel computing. *Journal of the American Society for Information Science* 49(3) (1998), 206.

[10] Sh. Chen, YC. Tu, and Y. Xia. 2011. Performance analysis of a dual-tree algorithm for computing spatial distance histograms. *VLDB* 20(4) (2011), 471–494.

[11] D. J. DeWitt and J. Gray. 1990. Parallel database systems: The future of database processing or a passing fad? *ACM SIGMOD Record* 19(4) (1990), 104–112.

[12] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *Data Engineering (ICDE).* 1086–1097.

[13] T. Foley and J. Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS on Graphics hardware.* 15–22.

[14] Z. Fu, X. Sun, Q. Liu, L. Zhou, and J. Shu. 2015. Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing. *IEICE Transactions on Communications* 98(1) (2015), 190–200.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. 2004. Fast computation of database operations using graphics processors. In *ACM SIGMOD.* 215–226.

[16] G. Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25(2) (1993), 73–169.

[17] Samet H. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16(2) (1984), 187–260.

[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. 2008. Relational joins on graphics processors. In *ACM SIGMOD.* 511–524.

[19] B. Hess, C. Kutzner, Van Der Spoel D, and Lindahl E. 2008. GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation* 4(3) (2008), 435–447.

[20] E. G. Hoel and H. Samet. 1994. Performance of data-parallel spatial operations. In *VLDB.* 156–167.

[21] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. 2007. Interactive kd tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games.* 167–174.

[22] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. 2011. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics* 17(4) (2011), 466–474.

[23] S. Ilarri, E. Mena, and A. Illarramendi. 2010. Location-dependent query processing: Where we are and where we are heading. *ACM Computing Surveys (CSUR)* 42(3) (2010), 12.

[24] Gluck J and A. Danner. 2014. Fast GPGPU Based Quadtree Construction. *Dep. C. Sc. Carnegie Mellon University* (2014).

[25] Wang K, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB* 7(11) (2014), 1011–1022.

[26] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31(5) (2011), 7–17.

[27] M. Kelly, A. Breslow, and A Kelly. 2011. Quad-tree construction on the gpu: A hybrid cpu-gpu approach. *Retrieved June13* (2011).

[28] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast bvh construction on gpus. In *Computer Graphics Forum*, Vol. 28(2). 375–384.

[29] H. Li, D. Yu, A. Kumar, and YC. Tu. 2014. Performance modeling in CUDA streams - A means for high-throughput data processing. In *Big Data, IEEE.* 301–310.

[30] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. 2005. Continuous query processing of spatio-temporal data streams in place. *GeoInformatica* 9(4) (2005), 343–365.

[31] O. Oguzcan, D. Funda, and G. Ugur. 2013. Dynamic point-region quadtrees for particle simulations. *Information Sciences* 218 (2013), 133–145.

[32] P. Rigaux, M. Scholl, and A. Voisard. 2001. *Spatial databases: with application to GIS.* Morgan Kaufmann.

[33] R.Rui and YC. Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management.* 17.

[34] H. Samet. 1988. An overview of quadtrees, octrees, and related hierarchical data structures. *NATO ASI Series* 40 (1988), 51–68.

[35] S. Tanimoto and T. Pavlidis. 1975. A hierarchical data structure for picture processing. *Computer graphics and image processing* 4(2) (1975), 104–119.

[36] Y. Theodoridis. 2003. Ten benchmark database queries for location-based services. *Comput. J.* 46(6) (2003), 713–725.

[37] H. Turtle and J. Flood. 1995. Query evaluation: strategies and optimizations. *Information Processing & Management* 31(6) (1995), 831–850.

[38] A. N. Wilschut and P. G. Apers. 1993. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases* 1(1) (1993), 103–128.

[39] S. You, J. Zhang, and Le. Gruenwald. 2013. Parallel spatial query processing on gpus using r-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data.* 23–31.

[40] B. Yu, H. Kim, W. Choi, and D. Kwon. 2011. Parallel range query processing on r-tree with graphics processing unit. In *Dependable, Autonomic and Secure Computing (DASC),IEEE.* 1235–1242.

[41] J. Zhang, S. You, and L. Gruenwald. 2010. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. In *Proceedings of the 18th SIGSPATIAL.* 450–453.

[42] K. Zhou, Q. Hou, R. Wang, and B. Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27(5) (2008), 126.