G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing

Zhila-Nouri Lewis and Yi-Cheng Tu

***Abstract*—Support for efficient spatial data storage and retrieval have become a vital component in almost all spatial database systems. While GPUs have become a mainstream platform for high-throughput data processing in recent years, exploiting the massively parallel processing power of GPUs is non-trivial. Current approaches that parallelize one query at a time have low work efficiency and cannot make good use of GPU resources. On the other hand, many spatial database applications are busy systems in which a large number of queries arrive simultaneously. In this paper, we present a comprehensive framework named G-PICS for parallel processing of a large number of spatial queries on GPUs. G-PICS encapsulates efficient parallel algorithms for constructing a variety of spatial trees with different space partitioning methods. G-PICS also provides highly optimized programs for processing major spatial query types, and such programs can be accessed via an API that could be further extended to implement user-defined algorithms. While support for dynamic data inputs is missing in existing work, G-PICS implements efficient parallel algorithms for bulk updates of data. Furthermore, G-PICS is designed to work in a Multi-GPU environment to support datasets beyond the size of a single GPU's global memory. Empirical evaluation of G-PICS shows significant performance improvement over the state-of-the-art GPU and parallel CPU-based spatial query processing systems. In particular, G-PICS achieves double-digit speedup over such systems in tree construction (up to 53X), and query processing (up to 80X). Moreover, tree update procedure outperforms the tree construction from scratch under different levels of data movement.**

I. Introduction

Spatio-temporal data has become a critical element in a broad range of applications such as geographic information systems, mobile computing, scientific computing, and astrophysics. Due to the high data volume and large query quantities, support for efficient spatial data storage and query processing has become a vital component in such systems. Popular spatial queries are spatial point search, range search, within-distance search, and k-nearest neighbors (kNNs) [1], [2]. Previous work has also demonstrated the great potential of parallel computing in achieving high performance query processing [3], [4]. However, if parallelism is adopted without spatial data indexing in query processing, the performance gain obtained will fade away quickly as data size increases [5], [6].

Over the last decade, many-core hardware has been widely used to speed up high-performance computing (HPC) applications. Among them, Graphical Processing Units (GPUs) have become a mainstream platform [7]. Spatial query processing on GPUs has also attracted much attention from the research community. Related work in this topic [8]–[10] focuses on parallelizing **one** search query at a time on GPUs. In [8], a GPU-based spatial index called STIG (Spatio-Temporal Indexing using GPUs) based on kd-tree is presented. In [10], a framework called GAT (GPU-accelerated Framework for Processing Trajectory Queries) is developed to support processing trajectory range queries and top-k similarity queries on GPUs. GAT is based on a quadtree-like index and cell-level trajectory representations. In [9], another variation of quadtree called Scout is developed to support spatio-temporal data visualization on GPUs. With plausible innovations, the above systems successfully demonstrated the potential of GPU-based spatial indexing, and also generated abundant opportunities for further research. This paper aims at a more comprehensive spatial indexing framework with even better performance and support of functionalities beyond query processing. In particular, we address the following issues.

a) High Performance in Tree Construction and Query Processing: In all the aforementioned work, a spatial tree consists of intermediate nodes, and a set of leaf blocks to store the spatial data records in consecutive memory locations. A two-step spatial query processing strategy is adopted in such work: (1) all leaf blocks are searched in a brute-force manner to identify those that satisfy the search conditions; and (2) all data points in the identified leaf blocks are examined to determine the final results. It is not easy to achieve a high degree of parallelism in the first step using traditional logarithmic tree search, especially when higher levels of the tree are visited. Hence, they adapt a *data parallel* solution for the first step on GPU, in which all the leaf blocks are transferred to GPU and scanned in a brute-force manner. However, by scanning all leaf nodes, such an approach is inefficient as it literally changes the total amount of work (*work efficiency*) from logarithmic to linear (Figure 1). Although they take 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 GAT, to achieve a logarithmic work efficiency, the first step is done on CPU by using a quadtree-based filtering method. Then, only the leaf blocks identified in the first step are transferred to GPU, and the second step is parallelized on GPU. Although the overall query processing performance is improved in GAT, it still suffers from the overhead caused by transferring the intersecting leaf blocks to GPU global memory thus has much room for improvement.

b) Handling Data Updates: An essential element that is missing from existing work is the support of data updates. In such work, the tree is constructed in host memory and transferred to GPU's global memory. In large datasets, building a tree is costly, and furthermore, the overhead of transferring data from CPU to GPU is significant. For static data, it is not an essential issue as tree construction and transferring is a one-

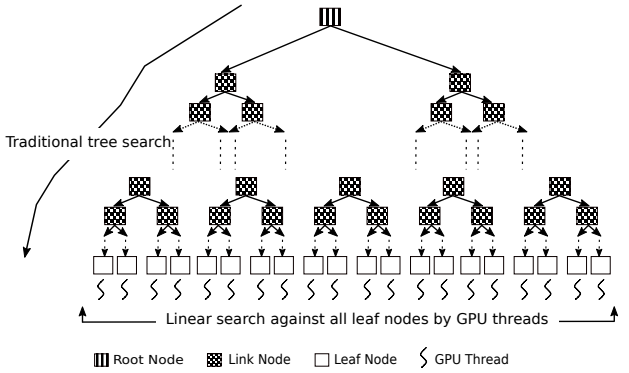

Fig. 1: Traditional tree search versus parallel linear search against all leaf nodes

time cost. However, almost all location-based services involve dynamic data. Without explicitly handling tree updates as in existing work, the tree will have to be rebuilt and transferred to the GPU every time there is update of data.

c) Multi-GPU Support: With today's data-intensive applications, efficient support for datasets that cannot fit into the GPU's global memory is necessary. To address that, GAT [10] uses a memory allocation table (MAT) to keep track of the leaf blocks residing in global memory. Therefore, before a query is launched on GPU, the MAT is first checked to see if queries' intersecting leaf blocks are in global memory. If not, such blocks have to be copied to global memory before query processing. In case the global memory does not have enough capacity for new leaf blocks, following a LRU swapping strategy, some leaf blocks are swapped out from global memory to make capacity for new blocks. Therefore, each time the overhead of transferring data from host memory to GPU memory is added to query processing. Thus, an essential step towards developing high performance spatial query processing in large datasets, is to reduce such overhead.

d) Multi-Query Optimization: In existing approaches, by processing one query at a time, optimization opportunities among different queries in a workload are wasted. For example, in the second step of the search, since each query scans a list of leaf nodes to find their data records, the same data record can be accessed many times by different queries in a workload. Consequently, the program easily hits a performance ceiling due to congestion of global memory while other high performance resources are either insufficiently utilized or largely unused (e.g., shared memory). Another drawback of these approaches is that query processing cannot proceed without CPU intervention.

It is well-known that many location-based applications are busy systems with very high query arrival rate [11], [12]. For example, 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 [13]. Therefore, there are optimization opportunities in co-processing concurrent queries. In [14], GAT is extended to support processing more than one query at a time by parallelizing each individual input query using the solution introduced in GAT. However, in this approach the number of queries that can be run simultaneously is limited to those that their intersecting leaf blocks can fit in GPU global memory. Therefore, the degree of parallelism is low and this approach cannot be used in query processing systems with high query arrival rate.

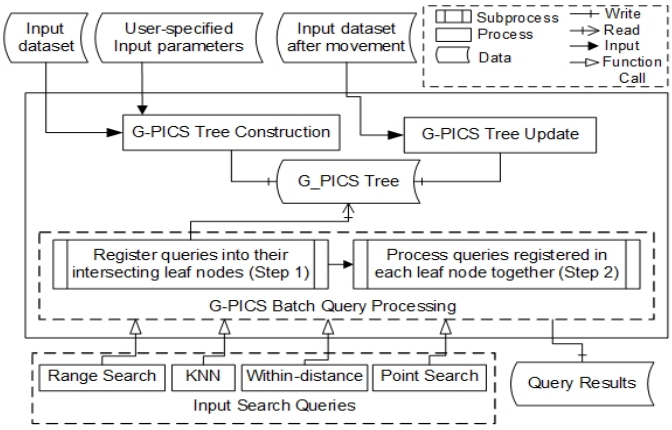

Fig. 2: Overview of G-PICS framework

A. Overview of Our Approach

In this paper, we present the **G-PICS** (GPU-based Parallel Indexing for Concurrent Spatial data processing) framework for high performance spatial data management and concurrent query processing. G-PICS is implemented as an extensible software package that supports various types of spatial trees under different hardware (GPU) specifications. Query processing approach in G-PICS bears logarithmic work efficiency for each query yet overcomes the problem of low parallelism. Therefore, instead of parallelizing a single tree-search operation, our strategy is to **parallelize multiple queries running concurrently**. Batched query processing, due to the effective sharing of computing resources, has been heavily studied in the database field [15]–[17], and large-scale web search on GPUs [18]. 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 query carries roughly the same work and is independent to others, it is easy to justify the use of parallel hardware.

G-PICS encapsulates all the key components for efficient parallel query processing within GPU with little CPU intervention. It includes highly efficient parallel algorithms for constructing a wide range of space partitioning trees based on user-input parameters. For example, users can choose to build trees with different node degrees and node partitioning method (e.g., space-driven or data driven). G-PICS provides APIs for processing major spatial queries including spatial point search, range search, within-distance search, and kNNs. Such APIs enable efficient development of more complex spatial data retrieval algorithms and applications. Figure 2 shows an overview of G-PICS framework.

G-PICS processes a group of spatial queries at a time, with each query assigned to a thread. Similar to existing work, query processing is accomplished in two steps. In the first step, following a traditional tree search approach, the leaf node(s) that contain the resulting data of each query are identified. However, instead of retrieving all the resulting

data, we only register each query to its corresponding leaf nodes. In the second step, following a *query-passive* design, the data in each leaf node is scanned only once and distributed 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. We conduct comprehensive experiments to validate the effectiveness of G-PICS. Our experimental results show performance boosts up to 80X (in both throughput and query response time) over best-known parallel GPU and parallel CPU-based spatial query processing systems. The G-PICS tree construction algorithms remarkably outperform the best-known parallel GPU-based algorithms – speedup up to 53X is reported. Moreover, tree update procedure outperforms the tree construction from scratch even under very high rate of data movement (up to 16X). G-PICS takes advantage of multiple GPU cards in a system to support large datasets with good scalability – by increasing the number of GPUs, we observe almost linear speedup.

B. Paper Organization

In Section II, we review other related work; in Section III, we introduce the tree construction algorithms developed in G-PICS; in Section IV, we present query processing algorithms in G-PICS; in Section V, we elaborate our algorithms to support data updates; in Section VI, we evaluate the performance of G-PICS, and in Section VII, we conclude the paper.

II. Other Related Work

In the past decade, GPUs are used to speed up computational tasks in many application domains. In the database field, GPUs are used to implement relational operators such as aggregates [19] and join [20]. Significant speedups were reported as the result of integrating GPUs with databases in processing spatial operations [21].

The needs of many applications require efficient data storage and retrieval via spatial indexes (space-partitioning trees) such as quadtree, and k-d tree [22]. Space-partitioning trees are hierarchical data structures in which a certain space is recursively divided into subregions. There are two groups of space-partitioning trees: data-driven, and space-driven. In the data-driven scheme, space partitioning is based on the input data, versus in space-driven the partitioning is only based on the space-specific rules. Selection of the appropriate space-partitioning tree for indexing the input data highly depends on the input data type and the search applications. For example, due to the highly balanced aspect ratio in space-driven partitioning trees, these trees are highly qualified for a large number of applications such as proximity searches, and range searches. In addition, space-driven partitioning trees are well-suited for indexing the uniformly-distributed or dynamic datasets [23]. Moreover, space-driven partitioning trees are shown to be highly compatible with parallel architecture especially GPUs [24], [25]. On the other hand, data-driven partitioning trees can be easily balanced (logarithmic depth) which is optimized for indexing highly skewed datasets.

Early work on GPU-based spatial indexes focused on computer graphics applications [26]–[28], with an aim for efficient triangle partitioning. In most existing work about spatial query processing on GPUs, spatial indexes are constructed on CPU and shipped to GPU for query processing. There are few work that focused on building the tree structure on GPUs. In [25] an algorithm for parallel construction of Bounding Volume Hierarchies (BVHs) on GPUs was developed. Point-Region (PR) quadtrees are simplified variations of BVHs in 2-D space. The idea introduced in [25] was later used in [29]–[31] to convert the PR quadtree construction on GPUs to an equivalent level-by-level bucket sort problem – quadtree nodes are considered as buckets and input data points are sorted into the buckets based on their locations at each level. K-d tree is a data-driven data structure, which is generally constructed level-by-level by dividing data points in each node into two partitions based on the median value of a chosen dimension. The dimensions in the data domain are chosen in a round-robin fashion at consecutive levels of the tree. In [32] a parallel algorithm for k-d tree construction on GPUs is presented, in which at each node the median value is selected by sorting the data points in the developing dimension using the CUDA Thrust library. However, sorting does more work than necessary for finding the median value [33], and performs poorly in dynamic datasets. There is no previous work targeted parallel data updates on spatial trees.

A preliminary version of this paper can be found in [34]. In this paper, we: (1) propose a new parallel data-driven partitioning tree construction algorithm on GPUs, (2) develop a more efficient algorithm for handling data updates comparing to the solution introduced in [34], and (3) extend the tree construction, query processing, and data update procedures to the Multi-GPU environment.

TABLE I: User-specified parameters for tree construction

Parameter	Meaning
N	total number of input data
NP	node degree (number of disjoint partitions)
MH	maximum tree height (number of levels)
MC	maximum number of data items in a node

III. Tree Construction in G-PICS

As discussed earlier, in previous work, the tree is usually constructed in host memory and then transferred to GPU for query processing. Due to the limited CPU computing power and memory bandwidth, the cost of building the tree on host memory is high. Moreover, the overhead of transferring the tree from host to GPU is significant - with its microsecond-level latency and 10GB/s-level bandwidth [35], 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 G-PICS, we provide support for building space-driven, and data-driven partitioning trees. Moreover, G-PICS supports tree construction for datasets whose sizes are bigger than a single GPU's global memory.

G-PICS is a general framework that supports any space partitioning tree that decomposes the space recursively to generate a fixed number of disjoint partitions each time. To

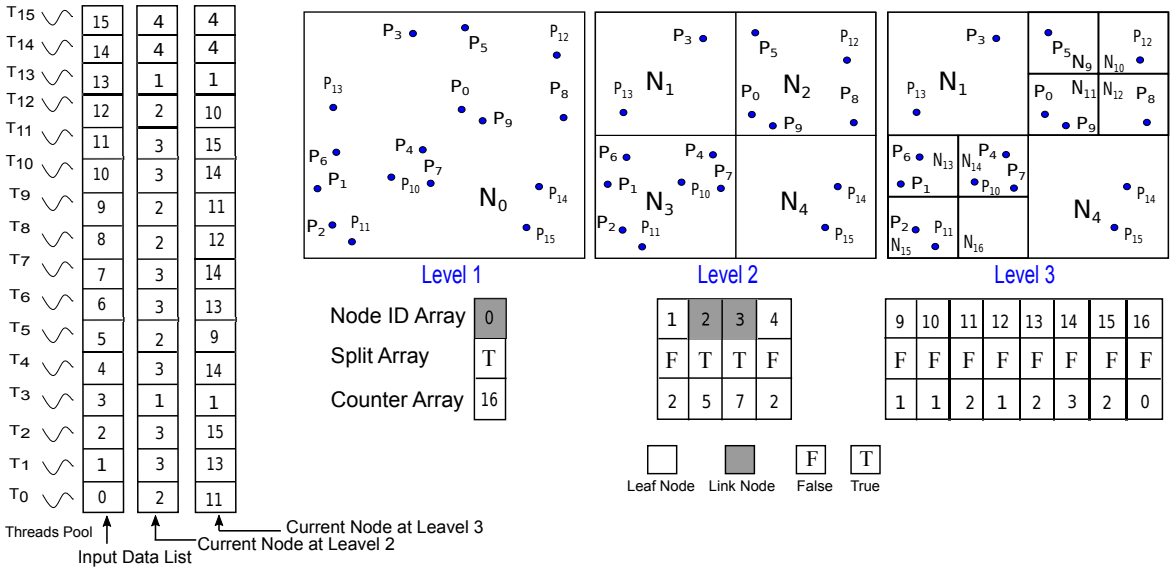

Fig. 3: An example of quadtree construction in G-PICS with $MC = MH = 3$. Auxiliary arrays with the length equal to the maximum number of nodes in a full tree are allocated on GPU and deleted when the tree construction is completed

construct a space partitioning tree, a user only needs to provide several parameters as inputs to G-PICS, and such parameters are summarized in Table I. If data points in a node exceeds MC, that node is partitioned into NP equal-sized child nodes. The decomposition continues until there is no more node to partition, or MH of the tree is reached [36] (stopping criteria). Nodes that got partitioned are link (internal) nodes, and others are leaf (data) nodes. G-PICS implements both space-driven and data-driven space partitioning methods to allow construction of a wide range of trees: with space-driven partitioning we can build point-region (PR) quadtree, region quadtree, PM quadtree, PMR quadtree, MX-PR quadtree, fixed grid, and point-region k-d tree [36], [37]. On the other hand, the data-driven partitioning allow users to build spatial k-d trees and other variants of Binary Space Partitioning trees. By changing the quantity NP, we will get trees with different degrees (e.g., degree 4 for quad-tree).

A. Space-driven Partitioning in Tree Construction

In space-driven partitioning trees, the space decomposition is totally independent to the dataset. For example, a PR quadtree is a type of trie, in which each link node has at most four children. If data points in a node exceeds MC, that node is partitioned into four equal-sized child nodes. The decomposition continues until the stopping criteria are met. There are unique challenges in the construction of space-driven partitioning trees on GPUs. First, to achieve high efficiency, our solution requires good utilization of GPU resources, especially the large number of cores. The traditional way for such is done by parallelizing the nodes' partitioning process level by level [29], [30]. Clearly, this approach suffers from low parallelism, especially when building top levels of the tree. Second, the total number of non-empty nodes in such trees is generally not known in advance. This is a major problem in GPUs as dynamic memory allocation on the thread level carries an extremely high overhead [20]. The easiest solution to tackle this problem, which was adapted in previous work

Algorithm 1: Space-driven Tree Construction Routine

Var: $splitNum$ (number of nodes to be partitioned) $\leftarrow 1$, C_{node} (array to keep current node for data points) $\leftarrow 0$, 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;

[29]–[31], is to nevertheless allocate memory for empty nodes. This results in inefficient use of (global) memory, which is of limited volume on GPUs, and becomes more crucial when dealing with skewed datasets. Finally, the main design goal of G-PICS trees is to allow efficient query processing. Hence, placing data points in a leaf node in consecutive memory locations is necessary, as it allows coalesced memory access in a data parallel program.

a) Overview of G-PICS Tree Construction: To address above challenges, we propose a top-down parallel algorithm on GPUs that achieves a high level of parallelism in the tree construction process. Furthermore, our approach avoids empty node expansion, and guarantees coalesced memory access in processing the data points in a leaf node.

G-PICS handles empty nodes 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 NP children. Let us use a full quadtree (Figure 3) as an example. The maximum number of nodes in such a tree (NP=4) with height of H can be calculated as $\sum_{i=0}^{i=(H-1)} 4^i = (4^H-1)/(4-1) = (4^H-1)/3$. Each node in a full quadtree has an ID, which is assigned based on its position in

Algorithm 2: Tree-Partitioning on GPU

Global Var: $Input$ (array of input data points)
Local Var: t (Thread id)
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: $L_{cnt} \leftarrow$ atomicAdd($Counter[C_{node}[Input[t]]]$, 1)
5: **if** $L_{cnt} == MC+1$ **then**
6: $Split[C_{node}[Input[t]]] \leftarrow$ True
7: **end if**
8: **end if**
9: **end for**

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.

The main idea behind G-PICS tree construction is a new parallel algorithm that maintains a high level of parallelism by novel workload assignment to GPU threads. Instead of binding a thread to a tree node which was adapted in previous work [29]–[31], each GPU thread is assigned to **one input data point**. By that, 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: Tree-Partitioning, Node-Creation, and Point-Insertion.

b) 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, saves the child node ID, and increments the counter of the number of points in the child node. Since such counts (i.e., counter array) are shared among all threads, we use *atomic instructions* to modify 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 the node should be further partitioned. 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 launched 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 second level of the tree is built. The kernel is relaunched with three new auxiliary arrays, the length of which corresponds to the number of the child nodes of only N_2 and N_3. Likewise, counter and split values of the nodes in this level are updated. This routine will continue until the stopping criteria are met. Our approach maintains a high level of parallelism by having N active threads at all times.

The Node-Creation kernel (Algorithm 3) is called to create

Algorithm 3: Node-Creation on GPU

Global Var: $leaf_{datalist}$ (array to store leaf nodes data)
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 point memory in $leaf_{datalist}$
5: **end if**
6: **end for**

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**

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 active threads as the number of non-empty nodes, each thread creates a non-empty node. While building nodes, memory for each leaf node's data list is allocated in consecutive memory locations. In Figure 3, the total number of non-empty nodes is 12 (while the full quadtree has 21 nodes).

The Point-Insertion kernel (Algorithm 4) is called to insert the input data points to the tree. 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.

c) Cost Modeling: The total cost of tree construction can be evaluated as follows:

$$C = C_T + C_I + C_P \quad (1)$$

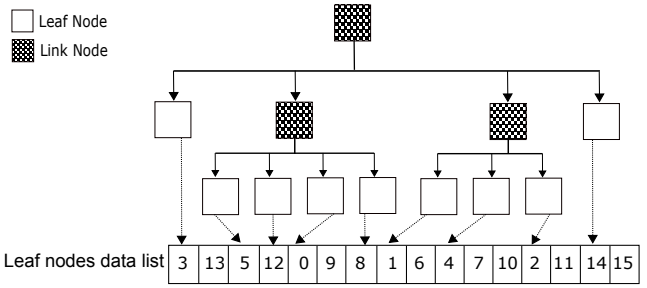

Fig. 4: Final quadtree built based on the data inputs in Figure 3

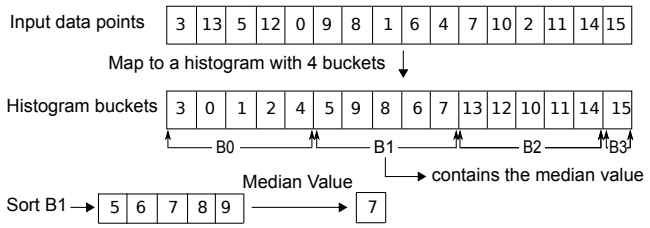

Fig. 5: An example of finding the median value via histogramming

where C_T, C_I, and C_P are the costs of Tree-Partitioning, Node-Creation, and Point-Insertion, respectively. Let N be the number of data points, and n the number of tree nodes, then, $C_T = O(N \log N)$, $C_I = O(n)$, and $C_P = O(N)$. Although C_T is of higher complexity, it is not the main bottleneck in practice. Instead, although C_P has a linear complexity, it is the dominating cost in tree construction. This is because Point-Insertion requires concurrent writes into the leaf node's data list, and this should be done via *atomic instructions* to ensure consistency. Atomic instructions are known to bear a large overhead. For example, our experiments show that on average 60 to 70 percent of the tree construction time is spent on the Point-Insertion kernel.

B. Data-driven Space Partitioning

Space decomposition is done based on the spatial distribution of the data to be indexed. A salient example is the spatial k-d tree, where each link node divides the space into two partitions (left and right subtrees) using a splitting hyperplane. Each link node in the tree is affiliated with one of the k-dimensions in the input dataset, with the hyperplane perpendicular to that dimension's axis. For instance in a node with "y" axis as splitting axis, all points in the left subtree have "y" values smaller than those in the right subtree. The splitting plane are chosen following a round-robin fashion. To ensure the tree is balanced, it is required that the pivot value for splitting be the median value of the coordinate values in the axis used to develop the splitting plane in each node. We face the following challenges in building data-driven trees. First, similar to space-driven partitioning trees, existing work [32] parallelizes the node's partitioning process by assigning each node to one GPU thread, and this causes low parallelism in building higher levels of a tree. Second, the median value for partitioning in each node is usually selected by sorting the data points based on the splitting axis [32]. Sorting is not an efficient way for finding the median value, as the entire sorted list is not required. Again, placing data points belonging to a leaf node in consecutive memory locations is required. Empty node expansions is not an issue in such trees since the partitioning is done based on the input data.

a) Overview of our approach: The key innovation is a parallel approach for finding the median value of a list of values without sorting. In order to find a median value in an unsorted list, data points in that list are organized into multiple continuous buckets using a range-based histogramming approach. The histogram is designed in a way to divide the input range of data lists into H histogram buckets. Each input data point ($Input$[i]) is assigned to one histogram bucket based on its location using a linear projection as follows:

$$\left\lfloor \frac{(H-1)}{(max-min)}(Input[i]-min) \right\rfloor \quad (2)$$

where max and min are the maximum and minimum value in the input list, respectively. Each bucket has a counter showing the number of points belonging to that bucket. By assigning the points to histogram buckets, the bucket that contains the median value can be determined based on the buckets' counter values. Thus, only that bucket will be sorted in order to find the median value. This approach eliminates extra candidates for sorting. In order to assign points to histogram buckets, each data point in a list is assigned to one GPU thread to find the bucket to which it belongs. Obviously, this approach achieves a high level of parallelism.

The main bottleneck lies in updating the counter values for each bucket by concurrent threads. To avoid race condition, atomic instructions have to be used for such updates yet atomic instructions are executed in a sequential manner. For that, we adopt an *output privatization* technique for outputting the histogram buckets' counters. Specifically, private copies of bucket counters are stored in on-chip cache called *shared memory* to be accessed by a subset of the threads. The private copies of the histogram are combined into the global histogram. After generating the final histogram output, the bucket that has the median value can be identified by reduction of counter values – median should divide the input list into two lists with equal size. Consequently, just the points in that bucket need to be sorted to find the median value. We implement two strategies for such sorting. If the number of points in that bucket are small enough to fit into shared memory, the sorting is done in shared memory using an efficient in-place fast shared memory sorting [38]. If the size of the input list is larger than shared memory, the sorting can be done using fast Radix sort libraries available in CUDA libraries such as Thrust or CUDPP. After sorting the points in the determined bucket, the median value in the input list is determined (Figure 5).

b) Tree Construction Routines: G-PICS k-d tree construction follows a level-by-level approach on GPUs to maximize the level of parallelism. At higher levels of the tree, every node in the developing level is assigned to a GPU thread, which launches another kernel using CUDA dynamic parallelism to perform the node partitioning. To partition a node, the median value in the developing dimension's axis is determined following the approach mentioned earlier. Having determined the median value, points can be organized into the sub-list (left and right) to which they belong. However, moving data points into sub-lists is not in-place, and consists a lot of atomic operations in global memory to find the next available pointer in each sub-list. To tackle such issue, in each GPU block, two private counters are defined in shared memory. Using the private counters, the total number of points belonging to each sub-list in each block is determined. Then, the private counters are used to increment the global counters in global memory. After moving data points to the sub-lists, nodes can be created. If the stopping criteria are met, a node is a leaf node; otherwise it is a link node and should get split

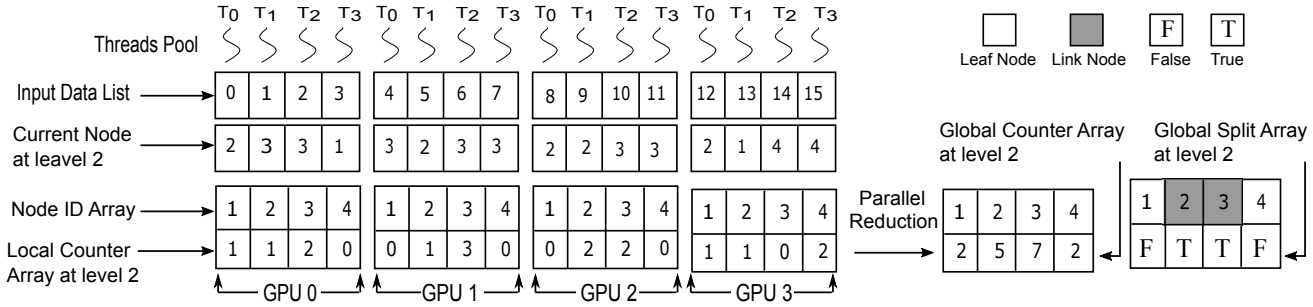

Fig. 6: Constructing the second level of the quadtree in Figure 3 using four GPUs

in building the next level of the tree. On the other hand, in the lower levels of the tree, the same approach is followed, however, each node that has to get partitioned is assigned to one GPU block for partitioning.

C. Tree Construction in Multiple GPUs

So far we assume that the entire tree plus all the data can fit into one GPU's global memory. For datasets with size beyond that, multiple GPUs should be used. The main challenge is to minimize data synchronization and communication among GPUs. For example, in order to determine whether a node should be split, summation of the auxiliary array values from all GPUs is required. Traditional wisdom adopts a *CPU master* approach [39] by using the CPU to perform such synchronization and communication. Specifically, all GPUs send their local copies of run-time parameters to the host CPU, and the CPU conducts a parallel reduction to get the global value and send the results back to the GPUs. However, this approach requires a lot of communications to transfer the parameters between all GPUs and CPU through PCI-E bus, which has low bandwidth and high latency. In addition, due to limited parallel computing resources, the reduction operation is much slower on the CPU. To overcome such limitations, we take a *GPU master* approach, in which one GPU becomes the central control, and all other GPUs send their local copies to the master GPU to conduct parallel reduction.

In addition, to minimize the communication cost, data transfer and in-core computation should be overlapped whenever it is possible. For that purpose, we adapt a GPU mechanism called **CUDA streams** to allow concurrent kernel function execution (Figure 7). Specifically, we simultaneously launch several CUDA streams containing GPU kernel calls via several OpenMP threads, and the data transferring is done using asynchronous direct GPU-To-GPU data transfer. In all the algorithms in this paper, whenever it is possible CUDA streams and asynchronous direct GPU-To-GPU data transfer are used to transfer data among GPUs.

a) Tree Construction Routines: Tree construction is still done in three steps. Since the input dataset size is bigger than one GPU's global memory, the input dataset should be split into multiple sets based on the number of available GPUs in a Multi-GPU cluster, and each set is assigned to one GPU. Each GPU in a work-set can work independently on finding enclosing node for points in its partition on the tree level being developed (Algorithm 2). However, in order to determine

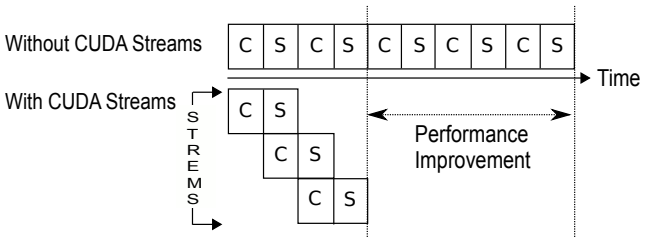

Fig. 7: Transferring Counter and Split arrays in Figure 3 via *CPU master* to four other GPUs comparing to *GPU master* approach with three CUDA streams, each containing two kernels on master GPU

whether the tree construction needs to proceed on building the next level of the tree, the nodes' data point counters should be updated based on the total sum of the counter values in all GPUs for that level. To this end, each GPU keeps a local copy of the node ID array, counter array, and split array. Then, after building each level, the corresponding global counter value for each node in that level is updated with sum of the values from each GPU using the GPU master approach (Figure 6).

The node creation kernel is launched on the master GPU (Algorithm 3). Since in such design all leaf node's data list cannot be stored in one GPU card, leaf node's data lists are allocated in consecutive memory locations on the available GPUs. Each leaf node on the master GPU keeps track of the physical memory location of its data list.

To insert the input data points to the data lists, each data point should be located in the same GPU global memory as its data list. Data points in each GPU that do not meet this condition, are grouped together based on the GPU to which they belong, and are copied to their corresponding GPU – using multiple CUDA streams and direct GPU-To-GPU data transfer in parallel. Then, the Point-Insertion kernel (Algorithm 4) is launched on each GPU to insert data points to their data lists. In the final quadtree structure, each leaf node points to the beginning of its data list in the leaf nodes data list array stored on one GPU global memory in a cluster.

D. Automated Code for Choosing the Partitioning Tree

We have presented parallel algorithms for building different types of tree structures within GPUs. It is clear that a suitable tree structure should be chosen based on the input dataset attributes and ultimate search applications on such dataset. For new users with G-PICS supported search applications, it may be challenging to choose a suitable tree structure. In our framework, we give two options to the user for choosing

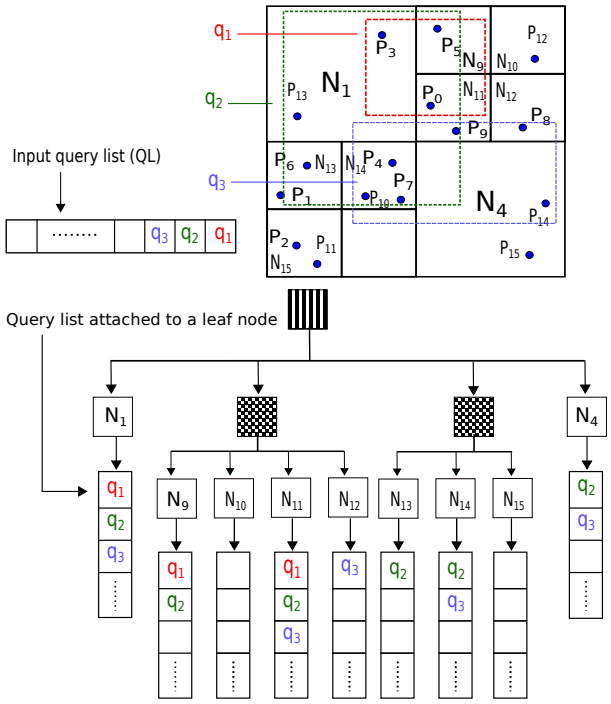

Fig. 8: An example of registering range search queries into the query lists of leaf nodes

the tree structure: (1) user can select the type of the tree to index the input dataset; (2) we encapsulate an algorithm in our framework that based on the user inputs determines the best tree structure for indexing the input dataset. To support the second option, our framework requires the following inputs: (1) estimation of input dataset size; (2) data distribution; (3) data type and number of dimensions; and (4) most common search applications on the input dataset. Based on such information, our framework selects the most appropriate indexing data structure, and number of GPUs for indexing the user input dataset.

IV. G-PICS Query Processing

As mentioned earlier, G-PICS supports the following types of spatial queries: (1) *spatial point search*, which retrieves data associated with a point in the data domain, (2) *range search*, which finds a set of data points intersect with a query shape object, (3) *within-distance search*, which retrieves objects within a specific distance from the search query, and (4) *k-nearest neighbors*, which retrieves k closest objects to a query point. G-PICS query processing algorithm is independent from the index structure (e.g., space-driven or data-driven).

A. Query Processing Algorithms in G-PICS

A typical spatial query is processed in two steps: (1) identification of leaf nodes satisfying the search conditions; and (2) examining the identified nodes to determine the final output data points. G-PICS is designed to process multiple spatial queries running concurrently. To this end, in the first step, using the traditional tree search is necessary to achieve logarithmic work efficiency. Recall that in the second step, reading the data records from GPU global memory is the main

Algorithm 5: Leaf-List-Processing on GPU

Local Var: b (Block id), t (Thread id), M (number of points in leaf[b]), lqL (query list for leaf[b]), sL (leaf[b] data list)

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

bottleneck, as the same data record can be accessed many times by different queries in a workload. To tackle these issues, G-PICS introduces a *push-based* paradigm for the second step of query processing. Specifically, a *query list* is attached to each leaf node for saving the list of queries intersecting that leaf node for processing. In the first step of query processing, queries intersecting with a leaf node are registered in the query list attached to that leaf node. In the second step, queries in each list are processed together to minimize accesses to global memory, and take advantage of the other available low-latency memories on GPUs. Two GPU kernels are designed to perform query processing: Leaf-Finding, and Leaf-List-Processing.

a) Step I - Query Registering: For each leaf node, we maintain a *query list*, which contains IDs of all queries whose outputting data can be found in that leaf node. In the Leaf-Finding kernel, each thread takes one query from QL, and finds the leaf node(s) that intersect with the query search key value or range. Then, registers that query to its intersecting leaf nodes. Figure 8 shows an example of such queries registering.

b) Step II - Leaf Node Processing: The Leaf-List-Processing kernel is launched with as many GPU blocks (i.e., a group of threads) as the number of leaf nodes. Then, each registered query in the query list is assigned to one thread in 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, if the number of registered queries is greater than one, all the data points belonging to the leaf node assigned to that GPU block are copied from global memory to shared memory. Shared memory is much faster than global memory - its access latency is about 28 clock cycles (versus global memory's 350 cycles) [40]. The copying from global memory to 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 shared memory, each active thread in that block takes one query from the query list attached to its corresponding leaf, calculates the Euclidean distance between that query point and all the points in that leaf node (located

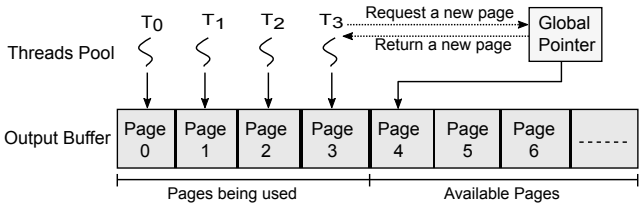

Fig. 9: An example of direct output buffer for GPU threads, showing Thread 3 acquiring a new page as its output buffer

in shared memory), and outputs those that satisfy the search criteria. This step is shown in Algorithm 5.

c) Query-Specific Implementation Details: The point search, range search, and within-distance search are implemented following the 2-step approach in a straightforward way: in the first step, a within-distance search retrieves the leaf nodes falling within a specific distance from the search key; a range search retrieves the leaf nodes intersecting with the query range; a point search retrieves the leaf node that contains the query search key. Then in the second step of the search, data lists in the identified leaf nodes from the first step are examined to output the final results. The kNNs 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 radius based on the input distribution and k. 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.

d) 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, or within-distance query. In CUDA, memory allocation with static size is preferred - in-thread dynamic memory allocation is possible but carries a huge performance penalty [20]. A typical solution is to run the same kernel twice: in the first round, output size is determined. In the second run, the same algorithm will be run again and output written to the memory allocated according to the size found in the first round. In G-PICS, we utilize an efficient solution introduced in our previous work [41], 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. In this design, an output buffer pool with a determined size is allocated. The allocated memory is divided into **pages** of a particular size. In order to record 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 the GP using the GPU atomic add operation. Using this solution, conflicts among threads is minimized because the GP is updated only when a page is completely filled (Figure 9).

e) Extensible Framework: G-PICS can be extended to support other spatial data retrieval algorithms and applications.

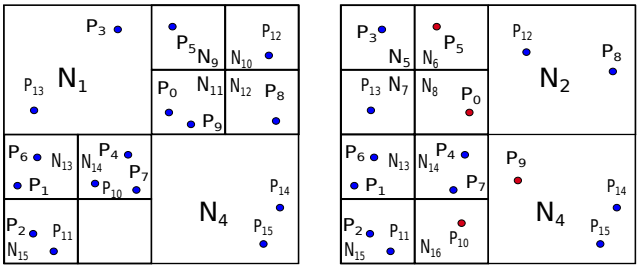

Fig. 10: Example of updating a quadtree with $MC = 3$ and $MH = 3$

For example, in our previous work [34], we extend G-PICS to support 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 this 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 all matching pairs within those leaf nodes are retrieved and outputted.

B. G-PICS Query Processing in Multiple GPUs

In Section III, we mentioned that the tree structure for large datasets is stored on one GPU's global memory (master GPU), and the leaf nodes' data are stored in multiple GPUs. Since the tree structure is stored on the master GPU, the first step of query processing is done on the master GPU. Then, the second step is parallelized using multiple GPUs. Query processing for such datasets is done in three steps as follows:

a) Step I - Query Registering: The Leaf-Finding kernel is done on the master GPU using the same approach mentioned in single GPU approach.

b) Step II - Copying query lists: In order to retrieve the final results, the query lists should be transferred to the corresponding GPUs where their intersecting leaf nodes' data lists are stored. The query list transferring is again done using multiple GPU streams and direct GPU-To-GPU transfer.

c) Step III - Leaf Node Processing: To process the registered queries in the query list of the leaf nodes on each GPU, the Leaf-List-Processing kernel (Algorithm 5) is launched with as many GPU blocks as the number of leaf nodes stored on that GPU to output the query results.

V. Tree Updates in G-PICS

G-PICS provides an efficient parallel update algorithm to support dynamic datasets on GPUs. Data movement may change the tree structure - it can be viewed as a deletion followed by an insertion. Both operations are costly because dynamic memory allocation at kernel runtime carries an extremely high overhead on GPUs, especially when there are group movements. At the end of each move, a data point can either stay in the same leaf node, or move into another node (red points in Figure 10). After processing all updates, the number of points in some leaf nodes may exceed MC. Consequently, if MH has not reached, the nodes should be partitioned, and points in them moved to their children (N_1). Alternatively, neighboring leaf nodes could lose data points and should be merged together (N_9, N_{10}, N_{11}, and N_{12}).

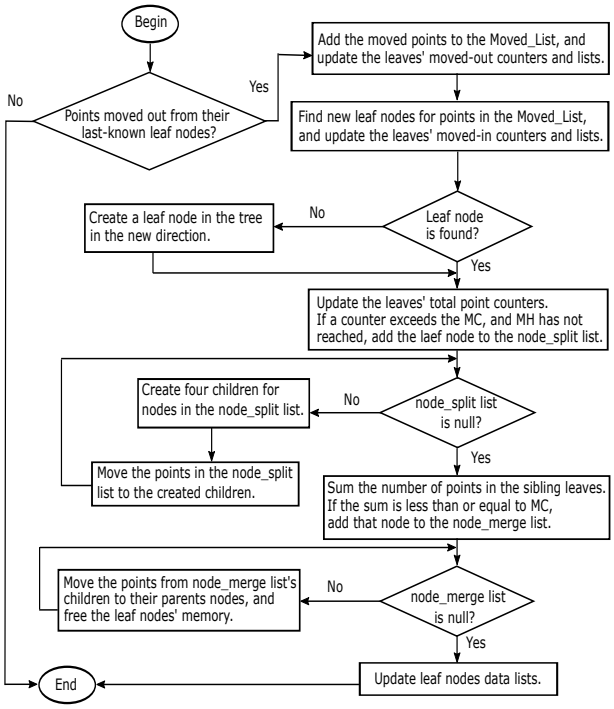

Fig. 11: Tree update procedure in G-PICS

Moreover, in space-driven partitioning G-PICS trees, empty nodes are not materialized; therefore, there may be cases that points move into an empty node (P_{10}) that needs to be materialized on-the-fly (N_{16}).

The easiest solution is to build the tree from scratch. However, this solution is inefficient in that all the data will be processed regardless of the amount of data movement. Therefore, our goal is to design an update procedure with running time **proportional** to the intensity of data movement.

A. Bulk Updates in G-PICS

We design a bottom-up parallel algorithm on GPUs to bulk update the tree under both data-driven and space-driven partitioning mechanisms. 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. If there is at least one movement, the tree structure should be updated accordingly. 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. Two counters, and two lists are added to each leaf node to keep track of points moved-in or moved-out during the update procedure. The work-flow of the entire tree update procedure is shown in Figure 11.

The Movement-Check kernel checks if each data point has moved out from its last-known leaf node. This kernel is launched with N threads. Each thread takes a point, and checks if the point has moved out from its last-known leaf node. In case of a movement, the corresponding thread adds the point to the list of moved data points ($Moved\_List$), and update the counters and lists associated with the last-known leaf node.

The Leaf-Finding kernel is called if $Moved\_List$ is not empty. This kernel is launched with as many threads as the number of points in $Moved\_List$. Each thread finds the new leaf node its assigned point moved in, and updates the relevant counters and lists associated with that leaf node. If updating the number of points in a leaf node makes it qualified for partitioning, that node is added to the *node_split* list. In case of moving to empty nodes in space-driven partitioning trees, new nodes are first created in parallel, and afterwards points are added to the newly-created nodes.

With a non-empty *node_split* list, the Node-Partitioning kernel is called to split the nodes in parallel. There are two groups of points that may belong to such nodes: points that previously belonged to those nodes and did not move out, and points from $Moved\_List$ that moved to those nodes. We call them $Candidate\_Points$. To maximize the efficiency in moving points from the partitioned nodes to the data list of the newly-built child nodes, only $Candidate\_Points$ are checked.

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 counters of sibling leaves. If the total number of points in those nodes become less than or equal to MC, they are added to the *node_merge* list. Siblings in this list are merged together, and their data points moved to their parent, which becomes a new leaf. The algorithm of moving points in these nodes is similar to the one in Node-Partitioning kernel.

To update the leaf nodes' data lists, a data point can be reinserted form scratch, as done in our previous work [34]. However, this approach does more work than necessary, especially under low data movement – any data movement would cause the entire data list rebuilt, which is the dominant cost. To increase efficiency, a memory allocation method for the data lists with the aforementioned page-based mechanism is designed as follows: first the leaf nodes that are affected by update procedure are identified. Then, only the data lists in those nodes are updated by calling Leaf-Lists-Update kernel, which assigns each leaf node to a GPU block. If in a leaf node the number of moved-in points is greater than moved-out points, then a new page is allocated to store the points moved into that node.

a) Cost Modeling: The total cost of update procedure can be expressed as:

$$C = C_M + C_L + C_V + C_D \tag{3}$$

where C_M is the cost of running Movement-Check, C_L the cost of Leaf-Finding, C_V the cost of updating the tree nodes, and C_D the cost of updating leaf nodes' data list. If α is the percent of points that moved out from their last-known leaf nodes, β is the percent of modified tree nodes, and γ is the percent of leaf nodes whose data lists have modified as the result of the update, we have $C_L = O(\alpha N \log N)$ and $C_V = O(\beta n)$ in both updating approaches, which are proportional to level of data movement. However, $C_D = O(\gamma\alpha N)$ in paging approach, and $C_D = O(N)$ non-paging approach. As discussed in Equation (1), C_D is the dominating cost. Since this cost is

proportional to the level of data movement in paging approach, it is expected that this approach shows better performance.

B. Tree Updates in Multiple GPUs

We also modified the bulk update procedure to support updates in multiple GPUs. To that end, the tree node structure (not including the data lists) replicates into all the available GPUs in the cluster. A local copy of counters and lists for keeping track of the movements are assigned to each copy on each GPU. Then, the new positions of points are sent to the GPUs to start the update procedure. Beside the kernels mentioned in the single GPU environment, two other kernels are designed to fulfill the task in Multi-GPU environment: Node-Counter-Updating, and Data-Points-Transferring.

The Movement-Check kernel is launched on each GPU with as many threads as the number of points assigned to that GPU followed by Leaf-Finding kernel call if $Moved\_List$ is not empty. To update the total number of points that moved in/out of a node, the local counter values are transferred to the master GPU for parallel reduction. Then, the Node-Counter-Updating kernel is called on the master GPU for updating the actual counter values in each node, and updating the *node_split* list.

The Node-Creation, Node-Partitioning and Node-Merging kernels are executed on the master GPU if the prerequisites for calling those kernels are satisfied. Then, finding the new location for points in the modified nodes is done using the GPUs where the data lists are stored.

Having finished all these steps, the data lists can be updated. As neighboring leaf nodes' data lists are most likely stored in the same GPU, data movement can be handled without communications between two GPUs. However, if there are points that move to a leaf node residing in another GPU, they will be transferred to corresponding GPUs using Data-Points-Transferring kernel before updating the leaf nodes data lists. Then, Leaf-Lists-Update kernel is called to update the data lists. This can be done using the page-based or reinsertion mechanism mentioned in the single-GPU approach.

VI. Experimental Evaluations

In this section, we present empirical evaluation of G-PICS performance. For that, we implemented a CUDA version of G-PICS including algorithms for processing the following search queries: window-based range, within-distance, kNNs, and point search. 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, equipped with four Nvidia Tesla P100 GPU cards with 16GB global memory in each card. All implementations in G-PICS and evaluation benchmarks 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 [42]. All the experiments are conducted over a real dataset [43] generated from a large-scale molecular simulation study of a lipid bi-layer system. [1] In the following

[1]The data contains 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.

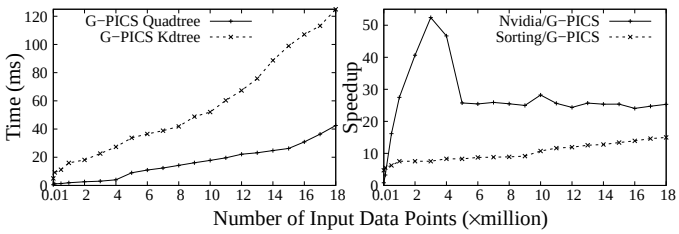

Fig. 12: Time for tree construction in G-PICS, and its speedup over Nvidia and kdtree-Sorting using CUDA Thrust library

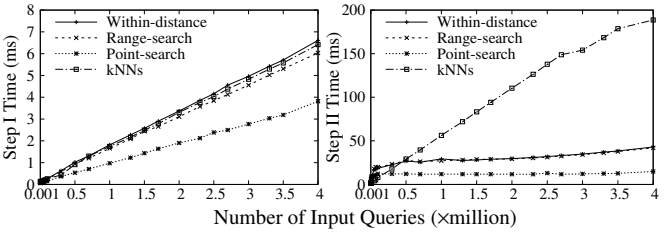

Fig. 13: Processing time for search queries in G-PICS

text, we report and analyze the absolute and relative total processing time (i.e., speedup) used for processing supported algorithms within G-PICS over various baseline programs.

A. Tree Construction in G-PICS

In such experiments, only the tree construction time is measured and compared. The parallel quadtree construction codes introduced in [29] and [30] are not publicly available. We implemented the algorithms following their descriptions. However, such algorithms showed very poor performance (G-PICS achieves more than 1000X speedup). Due to very low level of parallelism and inefficient memory use, the above academic code can hardly represent the state-of-the-art. Consequently, to have a more meaningful evaluation, we compare G-PICS tree construction with the parallel tree construction developed by Nvidia [31]. Nvidia code does not work for very large datasets, therefore, number of input data points in this experiment is limited to the ones that we can run using Nvidia. G-PICS kdtree construction performance is evaluated by comparing with parallel kdtree construction based on sorting all data points using the CUDA Thrust library [32].

Figure 12 shows the G-PICS quadtree and kdtree construction time, and speedup over comparative programs. As shown, G-PICS quadtree clearly outperforms the Nvidia ([31]) code (up to 53X) in all cases. By increasing the number of input data points, G-PICS speedup increases, and it remains constant at very large input datasets. While building the tree in G-PICS, if MH is reached, a leaf node could accommodate more points than MC. Under the same situation, the Nvidia code crashes. Moreover, G-PICS does not materialize the empty nodes, while Nvidia suffers from empty node expansion. Likewise, G-PICS kdtree construction beats the tree construction using sorting (up to 15X) in all cases.

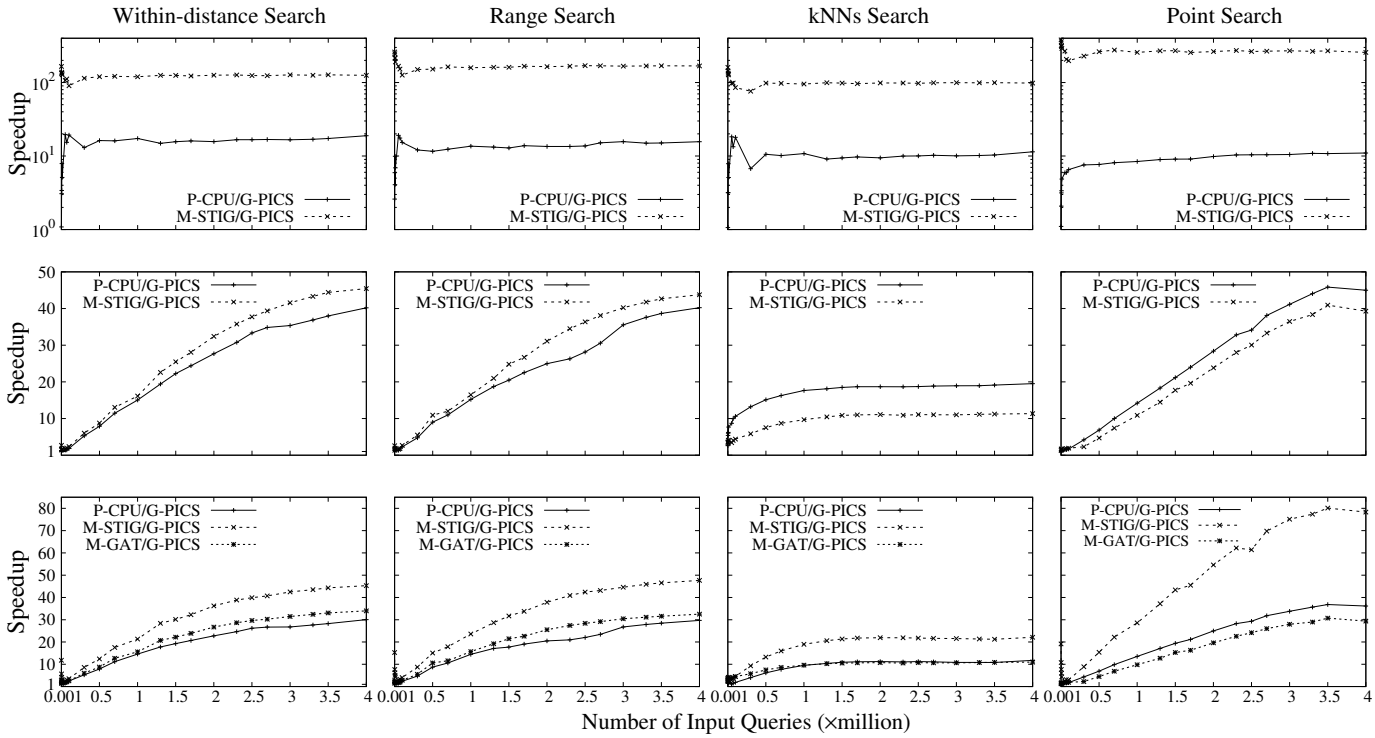

Fig. 14: Speedup (Top row: Step I only; Middle: Step II only; Bottom: Total time) of G-PICS over M-STIG, M-GAT, and P-CPU in processing 1,000 to 4,000,000 concurrent queries against a 16.5M-point database

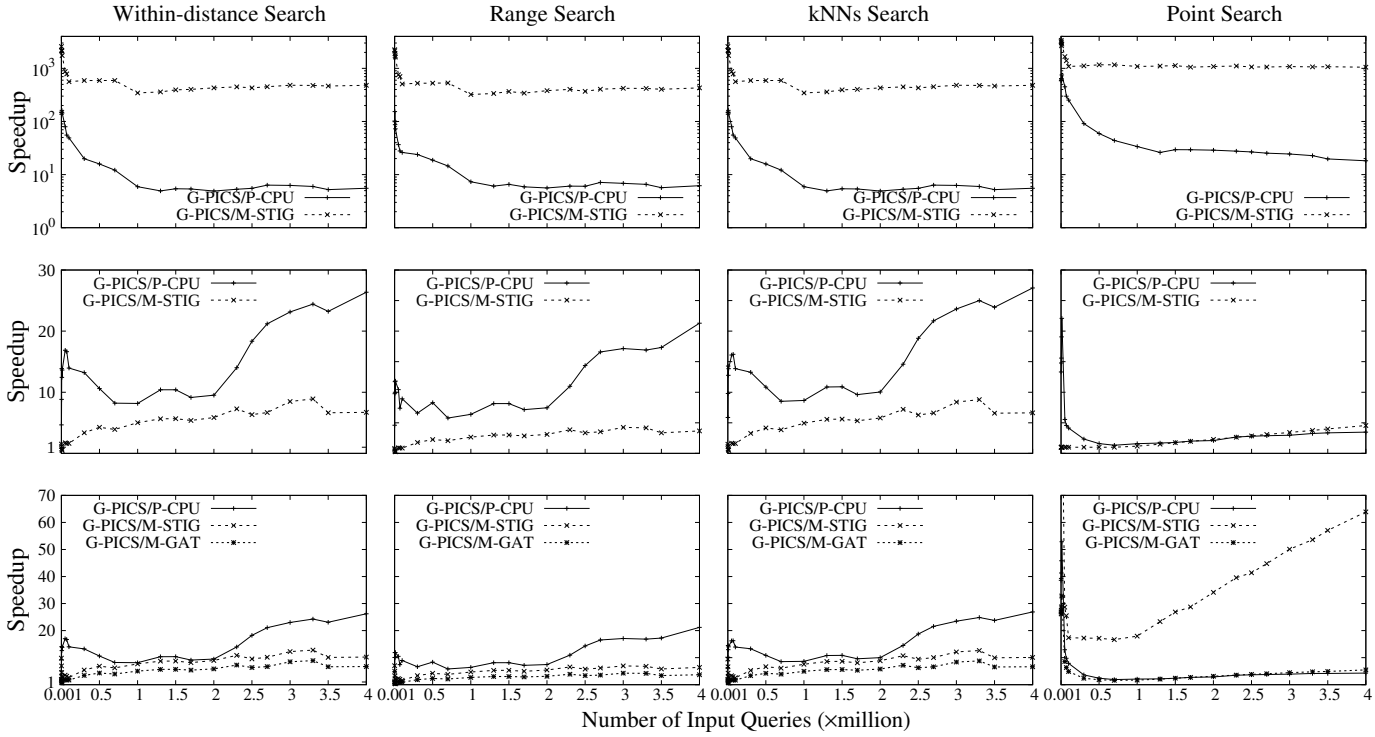

Fig. 15: Speedup (Top row: Step I only; Middle: Step II only; Bottom: Total time) of G-PICS over M-STIG, M-GAT, and P-CPU in processing 1,000 to 4,000,000 concurrent queries against a 256M-point database

B. Spatial Query Processing in G-PICS

STIG and GAT are not designed to support concurrent query processing for large number of queries. Queries have to be put into a queue and processed one after another. Therefore, for processing multiple queries using those solutions, input queries should be stored in a list. Then, while the list is not empty, one query is extracted from the list and parallelized on GPU. Consequently, the total processing time for each query is the sum of its processing time on GPU, and its waiting time in the list. For each query, the longer its index distance from the beginning of the list, the longer its total processing time. Consequently, the performance speedup achieves by parallelism is quickly offset by increasing the number of queries. For example, for processing 1000 queries using this approach G-PICS gains a performance speedup of 5000X and 180X over STIG and GAT, respectively. Such speedup shows a dramatic upward trend by increasing the number of input queries – with 10k queries, G-PICS speedup reaches 15000X and 500X over STIG and GAT, respectively. In addition, in GAT, leaf node blocks are copied to global memory, when they are accessed for the first time, which increases the processing time –such time is not considered in the above speedup comparison. STIG and GAT are not designed to support concurrent query processing for large number of queries. Therefore, for processing multiple queries using those solutions, input queries should be stored in a list. Then, while the list is not empty, one query is extracted from the list and parallelized on GPU. Consequently, the total processing time for each query is the sum of its processing time on GPU, and its waiting time in the list. For each query, the longer its index distance from the beginning of the list, the longer its total processing time. Consequently, the performance speedup achieves by parallelism is quickly offset by increasing the number of queries. For example, for processing 10K queries using this approach G-PICS gains a performance speedup of 250X over STIG. Such speedup shows a dramatic upward trend by increasing the number of input queries – with 50k queries, G-PICS speedup over STIG reaches 1000X. In addition, in GAT, leaf node blocks are copied to global memory, when they are accessed for the first time, which increases the processing time. For a meaningful evaluation, we compare G-PICS with the following three baseline algorithms: (1) a parallel CPU algorithm (P-CPU) implemented based on OpenMP. Note that P-CPU is highly optimized, and performs a parallel traditional tree search in Step I of query processing to bear the logarithmic work efficiency. Additional techniques for improving the P-CPU performance using OpenMP are applied including: choosing the best thread affinity for the thread scheduler, best thread scheduling mode, and best number of active threads; (2) M-STIG and (3) M-GAT, which are task parallel GPU programs for processing multiple queries at a time developed following the descriptions in STIG [8] and GAT [14]. Specifically, in Step I of query processing, M-GAT performs a parallel tree search on the CPU. Therefore, M-GAT performs the same as P-CPU in Step I for processing multiple queries concurrently. Then, the list of intersecting leaf nodes for queries are transferred to GPU. In Step II of the search, M-GAT parallelizes each query separately on GPUs. Hence, M-GAT performs the same as M-STIG in Step II for processing multiple queries concurrently. As the query processing is done in two steps, the total running time and that in each step (Step I and II) are compared. Note that the total time is end-to-end query processing time. This includes the time to ship query list into and the query results out of the GPU. This allows a fair comparison to P-CPU, which does not have such CPU-to-GPU transmission costs. The experiments in this section are run under different data sizes ranging from 16.5M to 256M points, which are indexed by a quadtree with MC and MH equal to 1024 and 14, respectively. Due to space limit, we only present the results of 16.5M and 256M datasets simulating a regular and very large spatial database. The query processing performance is evaluated under different numbers of concurrent queries (up to 4M). Since the output size of a typical range search and within-distance search is unknown in advance, we use the buffer pool solution discussed in Section IV for outputting the query results in G-PICS, M-GAT and M-STIG.

The absolute processing time of queries in G-PICS is shown in Figure 13. Since kNNs in G-PICS is done using within-distance searches followed by k closest selections in the within-distance search results, the processing time of Step II in kNNs is the sum of the processing time to perform both of these operations. We implement the same selection kernel for finding kNNs in G-PICS and M-STIG. For all 4 types of queries in both Steps, the processing time increases linearly with the number of concurrent queries. However, the time for running Step II dwarfs that for Step I therefore contributes more to the total processing time. Figure 14 shows the performance speedup of G-PICS over P-CPU, M-GAT, and M-STIG in processing different search queries. The logarithmic tree search in Step I noticeably outperforms the brute-force leaf search under all circumstances (more than 100X average speedup). The performance speedup over M-GAT and P-CPU in Step I is less remarkable (up to 20X) comparing to those over M-STIG. Such performance boost over M-STIG is certainly in conformity with the advantage of logarithmic tree search. Since there is less computation involved in processing Step I of the point search queries (each query just intersects with one leaf node), the speedup reaches a very high level (up to 300X). Generally, Step II speedup of G-PICS is not as high as that in Step I (up to 46X). It starts with a small number under low concurrency, then increases linearly with the number of input queries, and levels off afterwards. Such an increase of speedup is the result of using shared memory for holding data – the savings caused by caching are higher with more queries. When the shared memory bandwidth is fully utilized, the speedup levels off. As Step II dominates, the trends of speedup in total running time are similar to those found in Step II – even by considering the cost of transferring outputs back to CPU, G-PICS outperforms P-CPU remarkably.

a) Performance under very large dataset: To study G-PICS performance in handling very large datasets, we generated a dataset of 256M data points following the same data distribution in the 18M dataset. Note that dataset with such a size is the largest our algorithm can handle in the

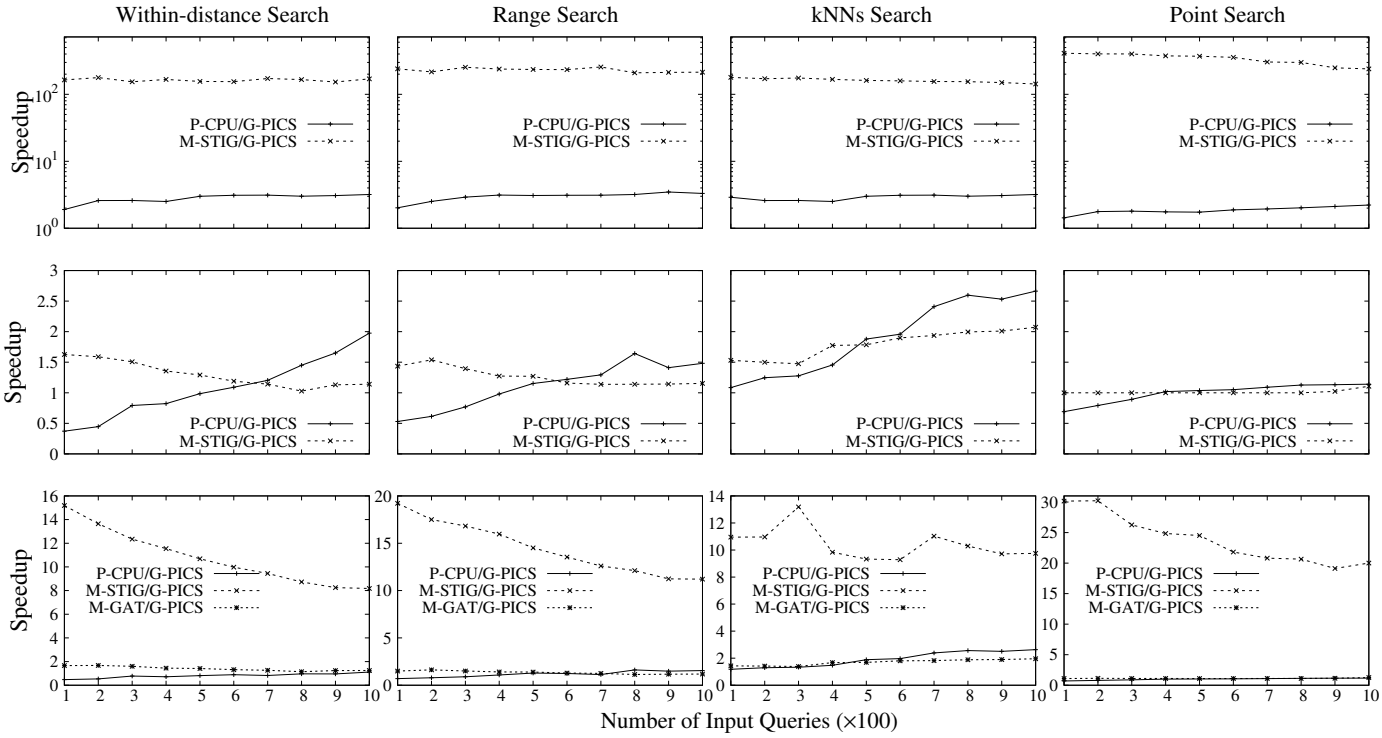

Fig. 16: Speedup in query processing time under low concurrency and data size 16.5M (Top row: Step I only; Middle: Step II only; Bottom: Total time) of G-PICS over M-STIG, M-GAT, and P-CPU

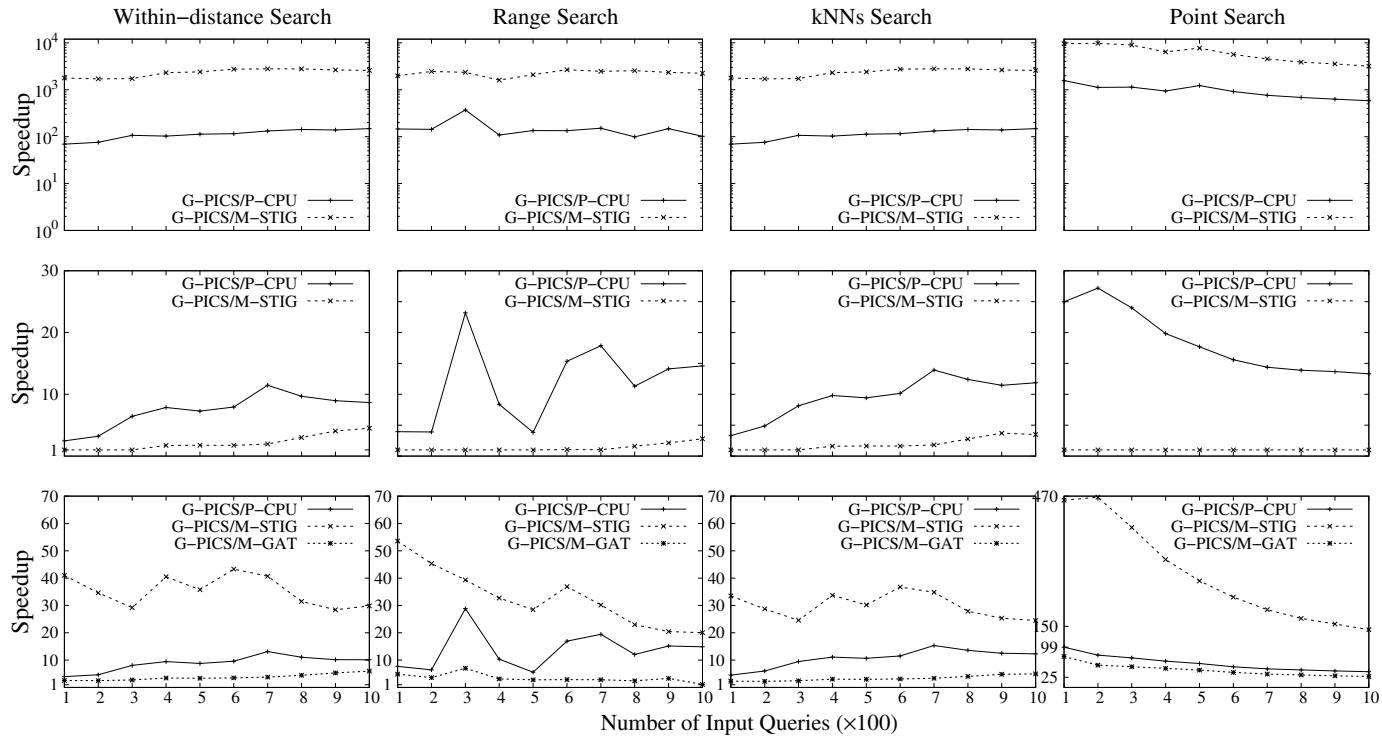

Fig. 17: Large dataset speedup under low concurrency for data size 256M (Top row: Step I only; Middle: Step II only; Bottom: Total time) of G-PICS over M-STIG, M-GAT, and P-CPU. Note that the total time speedup over M-STIG for the point search query is in a much higher range (140-460X) so we did not plot it

P100 card. We again indexed the data in a region quadtree constructed by G-PICS. The performance of G-PICS query processing algorithms are evaluated over the same baselines, and such results are presented in Figure 15. First, the speedup in Step I over M-STIG becomes much bigger as compared to the smaller dataset — it reaches 1,000X for the point search and about the same level for the other queries. This is understandable - the total work increases linearly with data size in M-STIG and only logarithmically in G-PICS. For the point search the difference is more obvious as each search intersects with only one leaf node. The performance speedup in this step over P-CPU followed the same trend as in the smaller dataset. Second, the speedup in Step II over M-STIG becomes smaller, reaching a level of 5-6X as compared to the 40X in Figure 14. The reason is because we used the same exact queries — when data gets bigger, average output size for each query increases accordingly. Therefore, the impact of sharing leaf nodes' data lists through shared memory in Step II becomes less significant comparing to M-STIG. On the other hand, the speedup over P-CPU is more significant in within-distance, range, and kNN search. This is because each query intersects multiple leaf nodes in their Step II of query processing, and cache miss is more common in the CPU as the processed data gets bigger. The Point search query, on contrary, just intersects with one leaf node thus P-CPU is more efficient. In addition, P-CPU follows the same design as G-PICS, therefore, fewer queries share the same leaf nodes data lists. Consequently, reading leaf nodes data lists becomes more costly in P-CPU comparing to G-PICS and M-STIG that take advantage of high bandwidth of GPU global memory. By looking at the total time speedup, again, it roughly follows the trends of Step II. In summary, G-PICS is still a few times more efficient than all the baselines.

b) Performance with the GeoLife Dataset:: We further evaluate the performance of G-PICS spatial query processing over GeoLife [44], which is a real-world trajectory dataset containing a total of 24.9M 2D coordinates. The speedup of G-PICS in spatial query processing using this dataset over the same baselines is shown in Figure 18. The results are very similar as those seen in Figure 14, with a speedup of G-PICS increases with concurrency level and reaches double-digit in processing all four types of queries. The only experiment that shows a different trend is for the kNN queries: the speedup increases more dramatically as compared to their counterparts in Figure 14, and reaching a higher value in the range of 20X to 40X. Since GeoLife dataset holds the trajectory points for trips, finding closer objects is much faster comparing to the other dataset.

c) Performance under low concurrency: Although G-PICS is designed for query processing systems with high query arrival rate, we still run experiments to evaluate the performance of G-PICS under low concurrency. Such results with the 16.5M-point dataset are shown in Figure 16. In Step I, the logarithmic tree search speedup in G-PICS over the brute-force leaf search in M-STIG is more remarkable under lower concurrency comparing to higher concurrency. Global accesses to the GPU global memory are cached in L2 cache inside the GPU. By increasing the number of queries, the hit rate for reading leaf nodes data through the cache increases. Therefore, this results in a slight improvement in brute-force leaf search accessing time in Step I. Considering Step II performance in G-PICS, if the number of registered queries in a leaf node is small, the performance speedup achieved using accessing leaf nodes' data lists through shared memory is not considerable – note that if there is one registered query in a leaf node, reading leaf nodes' data lists is done through global memory. Therefore, G-PICS speedup in Step II over M-STIG and M-GAT under low concurrency fluctuates based on query distribution in leaf nodes. However, due to its highly efficient Step I, G-PICS still outperforms M-STIG by at least 8X. On the other hand, having small number of threads, we are not able to fully utilize the GPU resources while that is not an issue for CPU code. Therefore, G-PICS Step I speedup over P-CPU and M-GAT is small. For the same reason, in Step II under very low concurrency, P-CPU performs better than G-PICS. However, by increasing the number of registered queries in each leaf node in Step II, the speedup of G-PICS over P-CPU increases.

Low concurrency results for the 256M-point dataset are shown in Figure 17. The effects of large data size on low concurrency performance are similar to those on high concurrency. However, Step I speedup becomes much bigger over M-STIG. Such speedup reaches to 10,000X for point search as each query at most intersects with one leaf node, and for each query a large number of leaf nodes should be searched in brute force manner. For Step II, the speedup is much higher over P-CPU, and is kept above 1X over M-STIG. Due to the great achieved speedup in Step I of point search, the total speedup for this query over M-STIG is much more significant (144-467X) comparing to the other queries and those under the 16.5M dataset (shown in Fig. 16). As data becomes large but the concurrency is low, the effects of Step II on total running time becomes much smaller therefore the total time speedup is only determined by that of Step I.

C. Tree Update Performance in G-PICS

To evaluate the performance of the tree update procedure, we change the positions of a certain percentage of the input data points to new randomly-generated positions. Then, the tree is updated accordingly. To measure the performance, the percent of the input data point that moved out from their last-known leaf nodes is captured. Then, the time it takes to update the tree is compared with that of building the tree from scratch using G-PICS tree construction code (we refer to it as *Scratch* hereafter). Figure 19 shows the speedup of the tree update algorithm using paging and non-paging approaches over Scratch. For both approaches, G-PICS outperforms Scratch. As we expected, with the increase of the intensity of data movement, the update performance decreases. Furthermore, the paging approach outperforms non-paging approach remarkably, even under very high level of movement. The experimental results are in compliance with update cost in Equation 3, which confirms the dominating cost in paging approach is proportional to the intensity of data movement.

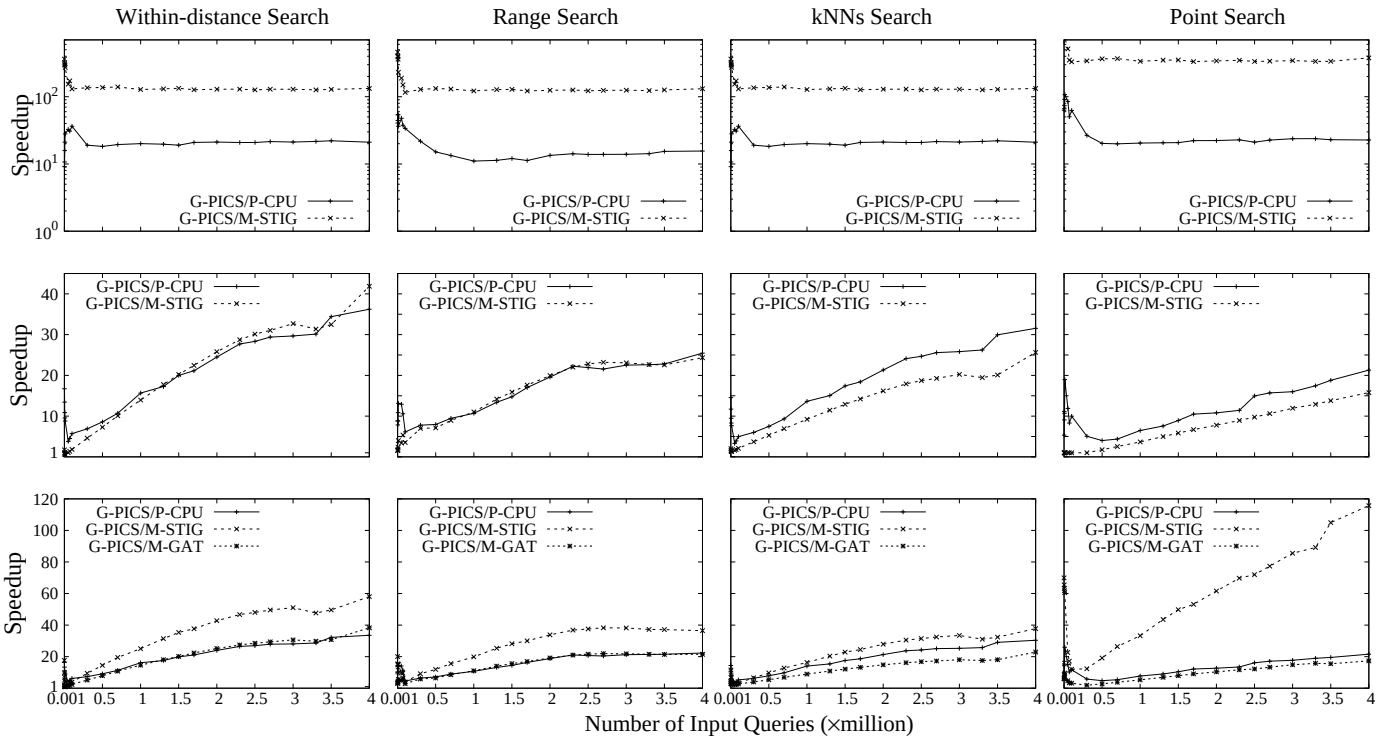

Fig. 18: Speedup (Top row: Step I only; Middle: Step II only; Bottom: Total time) of G-PICS over M-STIG, M-GAT, and P-CPU in processing 1,000 to 4,000,000 concurrent queries against the GeoLife dataset

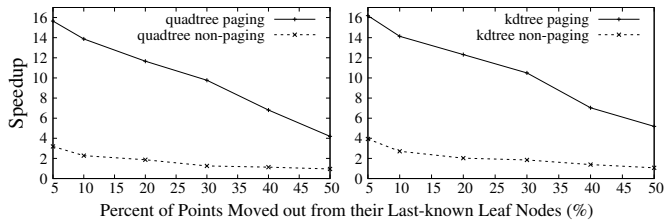

Fig. 19: G-PICS update performance over Scratch

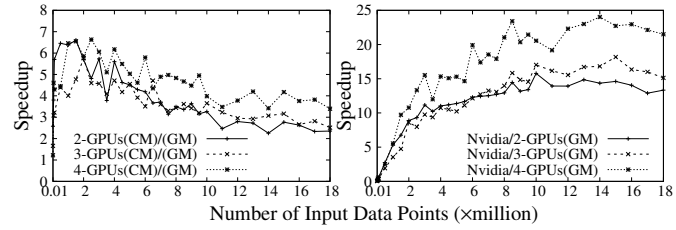

Fig. 20: Speedup of G-PICS tree construction using GM over using CM and Nvidia code in a Multi-GPU system

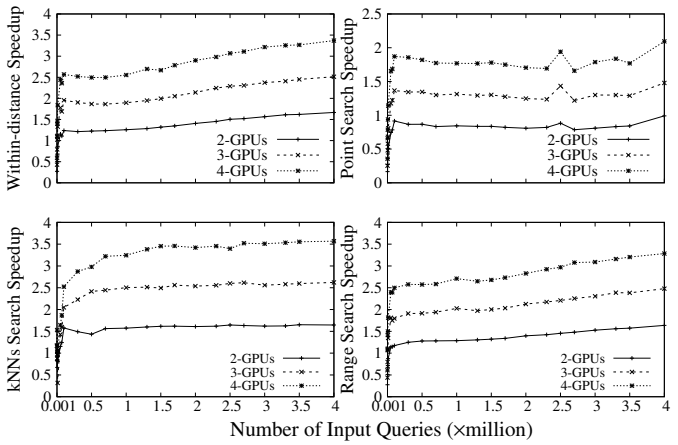

Fig. 21: G-PICS Multi-GPU search query processing Speedup over single GPU

D. G-PICS Performance in a Multi-GPU Environment

We evaluate our multi-GPU algorithms with a focus on performance scalability. For that purpose, we use a dataset that can fit into one GPU's global memory but distribute it to multiple GPUs. The experiments are conducted by using one to four GPUs.

1) Tree Construction: As discussed in Section III, we can build the tree in G-PICS via a GPU master (*GM*) or a CPU master (*CM*) approach. As expected, the *GM* approach shows much better performance than *CM* (Figure 20, left). GM shows at least a 2.5X speedup over CM under all numbers of GPUs. Figure 20 (right) shows the performance of *GM* over the Nvidia single GPU tree construction code [31]. With increasing sizes of input data, the performance speedup over Nvidia becomes more remarkable. In addition, the performance of G-PICS increases with more GPUs used, indicating good scalability.

2) Spatial Query Processing Performance: Results related to the performance of spatial query processing algorithms under multiple GPUs are shown in Figure 21, in which we plot the speedup of the algorithm running on 2 - 4 GPUs

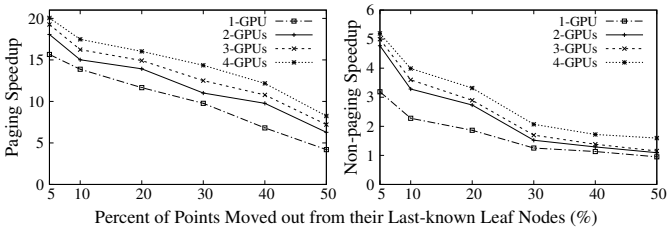

Fig. 22: G-PICS Multi-GPU update performance over Scratch

over that of a single GPU. In such experiments, we gain performance by distributing the computational workload to more GPUs while at the same time introducing data transmission overhead. A speedup higher than 1X is achieved when the gain overshadows the overhead. When two GPUs are used, the speedup is close to 1.5X for within-distance, kNNs, and range search queries while it goes slightly below 1X for point search. This is because the demand for in-core computation for the point search is low thus the communication cost dominates. However, when three or four GPUs are used, the performance increases significantly for all four query types. With four GPUs, the speedup reached 3.4X for within-distance, 3.5X for kNNs, 3.3X for range search, showing an (almost) linear scalability. For reasons mentioned above, the largest speedup for point search is only about 2X. In all query types, the speedup increases slightly or stays the same with the increase of the number of input queries.

3) Tree Update Performance: To measure the efficiency of multi-GPU tree updates in G-PICS, both paging and non-paging approaches are implemented and performance are evaluated against Scratch under the same set-up, e.g. tree update performance under four GPUs is measured against Scratch using four GPUs. According to Figure 22, the paging approach outperforms the non-paging approach using Multi-GPU with a similar trend as a single GPU. This is again in compliance with the update cost in Equation (3). Due to high communication cost among multiple GPUs, the multi-GPU update procedure is less scalable than query processing algorithms.

4) Scalability to larger data: We also demonstrate that by using multiple GPU cards, we are able to handle much larger datasets. Figure 23 shows such experiments for processing two million input queries in G-PICS using multiple GPU cards. As mentioned in Section IV, query processing in G-PICS using multiple GPU cards is done in three steps: (1) query registering on the master GPU, (2) copying query lists from the master GPU to other GPUs, and (3) leaf nodes' data lists processing and outputting the results on each GPU. Query registering is done on the master GPU using logarithmic tree search; therefore, by increasing the input dataset size, this cost increases in a logarithmic manner. Since the number of input queries is fixed, the cost of copying the query lists from the master GPU to other GPUs using multiple CUDA streams and direct GPU-To-GPU transfer does not change by increasing the input data size. However, the fewer the number of GPU cards, the higher the cost of transferring those lists to each GPU. This is because each GPU holds more leaf nodes' data lists. Considering the cost of leaf nodes' data lists processing, in point search queries, this cost does not increase a lot by increasing the input dataset – each query just intersects with one leaf node. Therefore, query processing time in processing point search queries follows a logarithmic increase by increasing the input dataset size. However, in other query processing types (range search, within-distance search, and kNNs), the same query range is used for all datasets. Therefore, increasing the input dataset size leads to increase in the volume of data to be processed, and accordingly increase in the output size for each query. Consequently, query processing time in those queries follows a linear increase by increasing the input dataset size. On the other hand, by adding more GPU cards, query processing time decreases linearly. Therefore, by adding more GPU cards, not only processing larger datasets is possible, but also linear speedup in terms of query processing performance is achieved. Thus, we can conclude that query processing algorithms within G-PICS scale very well across multiple GPU cards.

VII. Conclusions

In this paper, we advocate the adaptation of GPUs in spatial query processing, especially in applications dealing with concurrent queries over large input datasets. Existing work in this topic show low work efficiency and cannot make good use of GPU resources. To that end, we present a GPU-based Parallel Spatial Data Indexing framework for high performance spatial data management and concurrent query processing. G-PICS provides new tree construction algorithms on GPUs, which achieves a high level of parallelism and shows a performance boost of up to 53X over the best-known parallel GPU-based algorithms. Moreover, G-PICS introduces a 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 CPU-based and GPU spatial processing systems (up to 80X). In addition, G-PICS provides an efficient parallel update procedure on GPUs to support dynamic datasets which outperforms the tree construction from scratch by up to 16X. Furthermore, all algorithms within G-PICS work in Multi-GPU environments to support large datasets beyond the capacity of global memory.

References

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

[2] Y. Theodoridis, "Ten benchmark database queries for location-based services," *The Computer Journal*, vol. 46(6), pp. 713–725, 2003.

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

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

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

[6] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *SSDBM*, 2009, pp. 302–319.

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

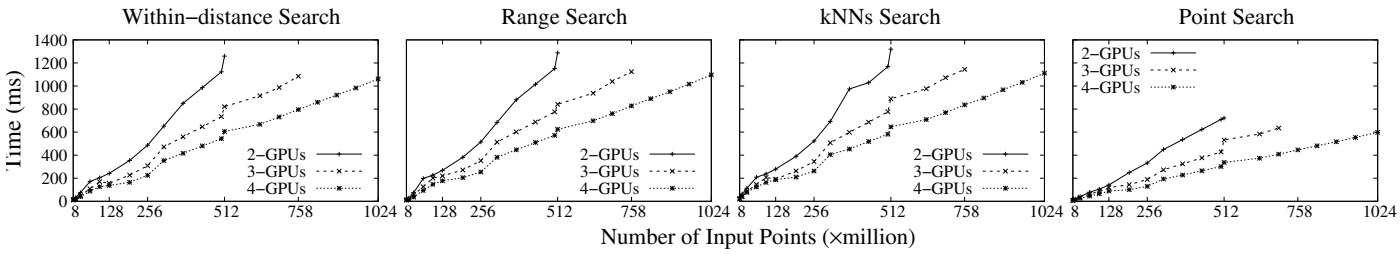

Fig. 23: G-PICS Multi-GPU query processing time for two million input queries

[8] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire, "A GPU-based index to support interactive spatio-temporal queries over historical data," in *ICDE*, 2016, pp. 1086–1097.

[9] H. Chavan and M. F. Mokbel, "Scout: A GPU-Aware system for interactive spatio-temporal data visualization," in *ACM International Conference on Management of Data*, 2017, pp. 1691–1694.

[10] B. Zhang, Y. Shen, Y. Zhu, and J. Yu, "A GPU-accelerated framework for processing trajectory queries," in *IEEE ICDE*, 2018, pp. 1037–1048.

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

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

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

[14] K. Dong, B. Zhang, Y. Shen, Y. Zhu, and J. Yu, "GAT: A unified GPU-accelerated framework for processing batch trajectory queries," *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[15] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems*, vol. 13(1), pp. 23–52, 1988.

[16] G. Giannikis, G. Alonso, and D. Kossmann, "SharedDB: killing one thousand queries with one stone," *VLDB Endowment*, vol. 5(6), pp. 526–537, 2012.

[17] S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez, "The datapath system: a data-centric analytic processing engine for large data warehouses," in *SIGMOD*, 2010, pp. 519–530.

[18] T. Kaldewey and A. D. Blas, "Large-scale GPU search," in *GPU Computing Gems Jade Edition*, 2012, pp. 3–14.

[19] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD*, 2004, pp. 215–226.

[20] B. He, K. Yang, R. Fang, M. Lu, and N. Govindaraju, "Relational joins on graphics processors," in *SIGMOD*, 2008, pp. 511–524.

[21] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi, "Hardware acceleration in commercial databases: A case study of spatial operations," in *VLDB*, vol. 30, 2004, pp. 1021–1032.

[22] A. W. G and I. I. F, "Sp-gist: An extensible database index for supporting space partitioning trees," *Journal of Intelligent Information Systems*, vol. 17(2), pp. 215–240, 2001.

[23] D. Eppstein, M. T. Goodrich, and J. Z. Sun, "The skip quadtree: a simple dynamic data structure for multidimensional data," in *Proceedings of the 21 symposium on Computational geometry*, 2005, pp. 296–305.

[24] E. G. Hoel and H. Samet, "Performance of data-parallel spatial operations," in *VLDB*, 1994, pp. 156–167.

[25] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," in *Computer Graphics Forum*, vol. 28(2), 2009, pp. 375–384.

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

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

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

[29] M. Kelly, A. Breslow, and A. Kelly, "Quad-tree construction on the GPU: A hybrid CPU-GPU approach," *Retrieved June13*, 2011.

[30] G. J and A. Danner, "Fast GPGPU based Quadtree construction," *Dep. C. Sc. Carnegie Mellon University*, 2014.

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

[32] L. Hu, S. Nooshabadi, and M. Ahmadi, "Massively parallel KD-tree construction and nearest neighbor search algorithms," in *IEEE ISCAS*, 2015, pp. 2752–2755.

[33] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens, "GPU multisplit: an extended study of a parallel algorithm," *TOPC*, vol. 4 (1), p. 2, 2017.

[34] Z. Nouri and Y. Tu, "GPU-Based parallel indexing for concurrent spatial query processing," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, 2018, pp. 23:1–23:12.

[35] "Low latency PCIe solutions for FPGA," Algorithm in Logic, Tech. Rep., 2017.

[36] H. Samet, "An overview of quadtrees, octrees, and related hierarchical data structures," *NATO ASI Series*, vol. 40, pp. 51–68, 1988.

[37] S. H, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16(2), pp. 187–260, 1984.

[38] M. Hongyu and G. Fangjin, "Simple sorting algorithm test based on CUDA," *arXiv preprint arXiv:1505.07605*, 2015.

[39] C. Raphael, R. Luiz, and S. Siang, "A multi-GPU algorithm for large-scale neuronal networks," *Concurrency and Computation: Practice and Experience*, vol. 23(6), pp. 556–572, 2011.

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

[41] R. Rui and Y. Tu, "Fast Equi-Join algorithms on GPUs: Design and implementation," in *Proceedings of SSDBM*, 2017, pp. 17:1–17:12.

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

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

[44] Z. Y, X. Xie, and M. W, "Geolife: A collaborative social networking service among user, location and trajectory." *IEEE Data Eng. Bulletin*, vol. 33(2), pp. 32–39, 2010.

Zhila-Nouri Lewis received a PhD degree in Computer Science and Engineering from University of South Florida in 2019. She received her Master's degree in Software Computer Engineering from University of Isfahan, Iran, in 2009, and her Bachelor's degree in Software Computer Engineering from Azad University of Najaf Abad, Iran, in 2006. Her current research is in big data, parallel and high performance computing.

Yi-Cheng Tu received a Bachelor's degree in horticulture from Beijing Agricultural University, China, and MS and PhD degrees in computer science from Purdue University. He is currently a full professor in the Department of Computer Science & Engineering at the University of South Florida. His research interest is in energy-efficient database systems, scientific data management, high performance computing and data stream management systems. He received a CAREER award from US National Science Foundation (NSF) in 2013.