Co-Processing Heterogeneous Parallel Index for Multi-Dimensional Datasets

Jinwoong Kim*, Beomseok Nam

School of Computer Science and Engineering
Ulsan National Institute of Science and Technology (UNIST), Korea

Abstract

We present a novel multi-dimensional range query co-processing scheme for the CPU and GPU. It has been reported that traversing hierarchical tree structures in parallel is inherently not efficient because of large branching factors. Besides, it is known that the recursive tree traversal algorithm required for multi-dimensional range queries is not well suited for the GPU architecture owing to its small shared memory.

In this paper, we propose co-processing range queries using both the CPU and GPU to make the most use of each architecture. In *Hybrid tree* that we present in this paper, we let CPU navigate the internal nodes of hierarchical tree structures and make GPU scan leaf nodes in a linear fashion using a massively large number of processing units. With the co-processing scheme, we can asynchronously leverage the strengths of each architecture. We also propose a novel dynamic GPU block scheduling algorithm for multiple range queries. In our scheduling algorithm, we consider the selection ratio of each query to determine the number of GPU blocks to launch. By assigning the right number of GPU blocks, we can significantly improve the query processing throughput for multiple concurrent queries. Our extensive experimental study shows that the proposed co-processing scheme shows up to 12x faster query response time than the state-of-the-art GPU tree traversal algorithm. We also show that

*Corresponding author
Email address: `jwkim@unist.ac.kr` (Jinwoong Kim)

our dynamic GPU block assignment algorithm improves the query processing throughput by up to 4x.

Keywords: GPU, Multi-dimensional Indexing, Query Co-processing, Parallel Indexing

2010 MSC: 68W10

1. Introduction

Scientific applications process truly large amounts of multi-dimensional datasets. To efficiently navigate such datasets, various multi-dimensional indexing structures, such as the R-tree [1], have been extensively studied in the past.

In the past decade, GPU has emerged as a cost-effective performance accelerator and various application domains, including medical image processing [2], computational chemistry [3], and particle physics [4], now leverage the massively parallel GPU architecture.

However, it has been reported that hierarchical multi-dimensional tree structures are inherently not well suited for parallel processing as their tree traversal paths are not deterministic because of large branching factors. Moreover, their irregular memory access patterns make it difficult to exploit massive parallelism [5]. On a GPU, recursive tree traversal often fails owing to its tiny run-time stack and small cache memory. Therefore, various brute-force linear scanning approaches instead of hierarchical tree-structured indexing have been employed in the literature [6, 7].

Recently, despite the difficulties of tree-structured indexing on the GPU, a few hierarchical tree-structured indexing techniques have been proposed for the GPU [8, 9, 10, 11, 12]. Massively Parallel Three-phase Scanning (MPTS) [12] and Massively Parallel Restart Scanning (MPRS) [9] are alternative tree traversal algorithms that scan multiple R-tree leaf nodes in a sequential fashion in order to avoid backtracking and minimize the warp divergence while effectively pruning a large portion of the tree nodes. They are shown to outperform CPU-based indexing and brute-force scanning methods on the GPU in terms of both query response time and query processing throughput. However, their performance gain mostly comes from leaf node scanning and the internal node traversal on the GPU suffers from the *warp-divergence* problem.

Another important drawback of the existing tree-structured indexing on the GPU is that the entire index must fit in GPU device memory. If an index is larger than the GPU device memory, some parts of the index must be managed in host memory. Recent advancements in GPU technologies have enabled GPU to directly access host memory via NVLink [13]. However, because the tree traversal path is non-deterministic, on-demand tree node fetching from host memory can significantly reduce query processing performance. To mask the tree node-fetching overhead, we can partition an index into sub-indexes and make the CPU process one of the sub-indexes while fetching another sub-index from host memory to GPU device memory.

In this work, we propose *Hybrid tree*, which partitions the R-tree into internal tree nodes and leaf nodes and stores them in CPU host memory and GPU device memory, respectively. The leaf nodes are stored as a single contiguous array, which we refer to as a *leaf array.* By statically partitioning the R-tree index into the CPU and GPU parts, we can concurrently utilize both the CPU and GPU and maximize the parallelism. For the internal tree nodes, the CPU achieves better performance than the GPU because it does not suffer from the warp divergence problem caused by conditional branches in the hierarchical tree structures. For the leaf nodes, the GPU scans a large number of leaf nodes in parallel according to the selection ratio of the range query. Since the GPU is known to be superior to the CPU for such parallel scanning.

A key challenge in this query *co-processing* is how to balance the workload between the CPU and GPU. First, we explore the opportunities for balancing the workload between the CPU and GPU for a single multi-dimensional query. This can be achieved by adjusting the length of the parallel scan in the leaf array. Next, we investigate how to co-process multiple concurrent multi-dimensional queries to improve the query processing throughput. With multi-core CPUs, multiple queries can concurrently traverse internal nodes for multiple queries;

however, they need to share the GPU blocks. Since the search paths of queries are usually different, each query accesses different parts of the leaf array and we need to schedule the GPU blocks for them. In this work, we present a dynamic GPU block scheduling algorithm for multiple query co-processing.

The contributions of this paper are summarized as follows.

- **Query co-processing on heterogeneous architectures**
  We propose a novel co-processing scheme for multi-dimensional range query. Our proposed algorithm asynchronously executes CPU and GPU computations and effectively overlaps the query processing time. By leveraging both the CPU and GPU, we can reduce the amount of brute-force scanning on the GPU and the number of internal nodes visited on the CPU.

- **Dynamic GPU block scheduling for multiple range queries**
  Balancing the workloads between the CPU and GPU is important in improving the query processing throughput and response time. To efficiently process multiple concurrent queries, we propose a dynamic GPU block scheduling algorithm that assigns more GPU blocks to the queries that can be rapidly processed by sequentially accessing a large number of leaf nodes.

The experimental results show that the proposed multi-dimensional range query co-processing scheme improves the query response time by up to 12x and the multiple query processing throughput by up to 4x.

The remainder of this paper is organized as follows. In section 2, we present previous related work as background. In section 3, we describe the details of our proposed heterogeneous co-processing scheme for multi-dimensional indexing. Subsequently, our experimental results and analysis of the performance are presented in section 4. In section 5, we conclude this paper.

2. Related Work

In the computer graphics community, various techniques have been proposed to overcome the tree recursion problem on the GPU [14, 15, 16, 17]. Foley et al. [14] proposed restarting tree traversal instead of backtracking. Hapala et al. [15] proposed enabling backtracking via auxiliary *parent link* pointers. Horn et al. [18] employed a small stack that leaks from the bottom when the fixed-sized stack becomes full. While these algorithms are designed for computer graphics applications and are proven to improve query processing throughput, they are not designed to improve the query response time of individual queries. In scientific applications, the number of concurrent queries is usually orders of magnitude smaller than the number of rays in computer graphics. Hence, such task-parallel stackless tree traversal algorithms are not sufficient in the scientific applications domain.

To improve the query response time, multiple processing units are required to cooperate in order to process a single query. The easiest form of data parallelism in indexing is brute-force scanning. Although brute-force scanning methods access the entire or a large portion of datasets, they have been shown to outperform the CPU-based tree-structured index by exploiting a large number of processing units and high memory bandwidth of the GPU [6, 7].

For a data-parallel tree-structured index, Kim et al. proposed FAST (Fast Architecture Sensitive Tree) [8]. To maximize the thread-level parallelism on the GPU, FAST rearranges a binary search tree into tree-structured blocks considering the cache line size, page size, and SIMD width. Each block of FAST is a parallel processing unit in a single streaming multiprocessor (SMP), which is similar to the tree node of n-ary trees, such as the B-tree. The block size of FAST is chosen to maximize the memory bandwidth of the GPU device memory. Unlike our work, FAST is designed for one-dimensional query, and does not consider co-processing.

Shahvarani et al. [10] proposed the HB+tree, similar to our work, which also utilizes both the CPU and GPU. In the HB+-tree, internal nodes are duplicated in the GPU device memory so that the GPU can concurrently process them in parallel. Unlike the HB+tree, we use the GPU for leaf node scanning instead of internal node traversal. Experimental results show that parallel leaf nodes scanning on the GPU is more effective in utilizing the high-memory bandwidth of GPUs. In addition, the HB+tree is designed for one-dimensional query, whereas we propose a co-processing scheme for multi-dimensional range query.

As for data-parallel multi-dimensional range query processing on the GPU, Luo et al. [19] proposed a parallel R-tree traversal algorithm, which employs an additional small queue in the shared memory of an SMP to store the bounding box overlap information. This approach is not scalable because the shared queue requires a locking mechanism for transaction semantics, which significantly reduces the performance.

The MPRS algorithm proposed by Kim et al. [9] is the state-of-the-art multi-dimensional indexing on the GPU. It traverses the hierarchical tree structures from root node to leaf nodes multiple times, as in the kd-restart algorithm. However, once it visits a leaf node, the MPRS algorithm scans some number of right sibling leaf nodes instead of backtracking to its parent node. The MPRS algorithm stops visiting right sibling leaf nodes when it visits a leaf node that does not contain any overlapping data points. Instead of accessing the right sibling leaf nodes further, it restarts the tree traversal from the root node. Similar to our work, MPRS avoids recursion and the GPU mostly scans leaf nodes sequentially. However, MPRS is not a query co-processing algorithm, which is the key contribution of this work.

3. Co-Processing Hybrid Tree in Parallel

3.1. Hybrid Tree

To co-process a multi-dimensional range query using both the CPU and GPU, we propose *Hybrid tree*, which partitions a multi-dimensional index into *upper tree* and *leaf array*, as shown in Figure 1. The upper tree is a traditional multi-dimensional tree-structured index that resides in the CPU host memory,

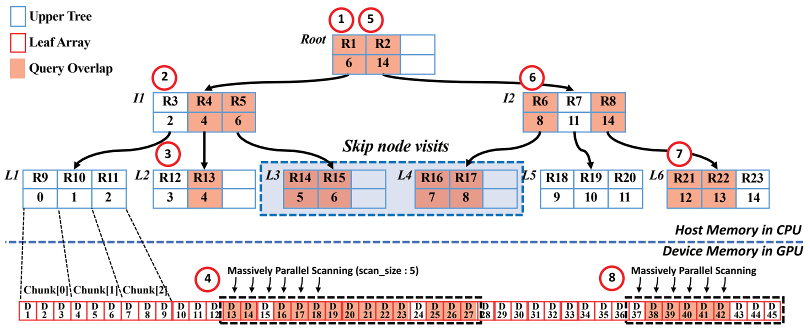


Figure 1: An example of a co-processing Hybrid tree with CPU and GPU: Leaf node scanning on the GPU (step 4) overlaps the next tree traversal on the CPU in time(steps 5-7)

and the leaf array is a contiguous memory block in the GPU device memory that comprises leaf nodes. For the upper tree, any multi-dimensional indexing structures, such as the R-tree, bounding volume hierarchy (BVH), or KDB-trees, can be employed.

The leaf array is logically partitioned into small chunks. Each chunk corresponds to a leaf node in a legacy R-tree, except that the chunks are stored adjacent to each other in a single large contiguous memory block, and the size of a chunk does not have to be equal to that of the internal tree nodes. For each leaf node, we build a minimum bounding box (MBB) and store it in the upper tree. As we increase the size of a leaf node, the number of leaf nodes decreases and the size of the upper tree decreases accordingly. In the legacy R-tree, the size of a tree node is determined by disk block size. However, since our Hybrid tree is an in-memory index, the size of each leaf node should be determined considering the relative processing power of the GPU and CPU. If the GPU is more powerful, more workload should be assigned to it. We can increase the workload of the GPU by increasing the size of a leaf node or can ask the GPU to process multiple leaf nodes. In each leaf node, we store $n \times k$ data points, where k is the number of data points that each GPU thread should process and n is the number of threads in a single GPU block.

The size of the MBB is a critical performance factor affecting the search performance of indexing structures. To reduce the size of the MBBs of leaf nodes, data points in the chunk array must be clustered. For this purpose, we employ a *Hilbert space-filling curve* [20] that assigns a one-dimensional value to each multi-dimensional point for maintaining good spatial locality. In other words, data points close to each other in multi-dimensional space are given similar Hilbert index values. Using the Hilbert index value, we sort the data points and store them in the leaf array. Sorting is an expensive operation; however, the GPU can accelerate sorting via various parallel sorting algorithms [21, 22].

Data layout on the GPU device memory is known to have a critical performance impact. A common access pattern into a multi-dimensional index is the comparison of a query's coordinate and the MBB coordinates in a particular dimension, which can be performed in a SIMD fashion. To reduce the number of device memory accesses, we store the sorted data points in the leaf array as a structure of arrays (SoA) layout shown as follows.

```
structure {
  float min[nDims*nDegrees]; // lower bounds
  float max[nDims*nDegrees]; // upper bounds
}
```

With this layout, the coordinates of the data points in the same dimension are contiguously stored. Hence, this minimizes the number of cache lines that are brought into the memory. Additionally, we can efficiently check the coordinates in a SIMD fashion. If we store the data points in an array of structures (AoS) layout,it may fetch as many cache lines as the number of data points in the worst case.

3.2. Co-Processing Hybrid Tree

In our query co-processing algorithm, we let the CPU process the upper tree while the GPU processes the leaf array. In traditional tree-structured indexing,

Algorithm 1 *Co-Processing Range Query with Hybrid Tree*

```
void Hybrid_RangeQueryProcessing(MBB query, Node root_node, int scan_size)
1: long $scan\_start \leftarrow 0$;
2: long $scan\_end \leftarrow 0$;
3: while $true$ do
4:     // traverse upper tree : ignore left sub-tree
5:     // whose leaf index is smaller than scan_end
6:     $scan\_start = TraverseUpperTreeOnCPU(query, root\_node, scan\_end)$;
7:     // no more node overlaps in upper tree
8:     if $scan\_start == 0$ then
9:         // terminate query
10:        $break$;
11:    end if
12:    // Start parallel scanning of leaf array on the GPU
13:    $ScanLeafArrayOnGPU(query, scan\_start, scan\_size)$;
14:    // We restart the tree traversal without waiting
15:    // for the GPU to finish parallel scanning.
16:    $scan\_end = (scan\_start + scan\_size)$;
17: end while
```

only one tree node is accessed at any level, i.e., we need to visit one of the child nodes at the next level or return to its parent node for backtracking.

In our novel co-processing algorithm described in Algorithm 1, we make GPU scan a certain number of leaf nodes in the leaf array. Brute-force exhaustive scanning poorly performs on the CPU because it has a small number of cores and memory bandwidth is low. However, the GPU provides higher memory bandwidth and it has more processing units than CPU. Hence, brute-force exhaustive parallel scanning on the GPU often performs better than on the CPU, which performs better when executing sophisticated and sequential algorithms. If an algorithm having a high branching factor is executed on the GPU, it poorly utilizes the high memory bandwidth and suffers from low computational power. But if simple brute-force algorithms are executed in the GPU, its low computational power is compensated by the high memory bandwidth and massive parallelism that benefit from a low branching factor.

While the GPU asynchronously processes the leaf array via brute-force exhaustive scanning, we make CPU restart the upper tree traversal from the root node once again. But this time, the traversal of the tree structure only involves visiting the leaf nodes that have not been visited. Since we know which part of the leaf array will be concurrently handled by the GPU, we can make CPU skip visiting the parent nodes of the leaf nodes that will be accessed by the GPU. This is achieved by adding the max *leaf node index* into the leaf array field in each internal tree node. In Figure 1, the numbers in the second row of each tree node are the max heap of the leaf node index. When we traverse the upper tree, we compare the leaf node index of each child node with the largest leaf node index of the leaf array to be accessed by the GPU (referred to as `scan_end`). By preventing each traversal from visiting tree nodes whose leaf node index is smaller than the `scan_end`, we can avoid revisiting leaf nodes.

To make the CPU and GPU effectively co-process the internal tree nodes and leaf nodes, respectively, the amount of workload in the CPU and GPU must be similar. In our co-processing scheme, the balance between the CPU and GPU workloads can be controlled by the `scan_size`. The scan size is the number of leaf nodes to be scanned by the GPU. In legacy multi-dimensional index traversal, we backtrack to a parent node after visiting its leaf node. However, in our co-processing algorithm, we make the GPU visit multiple sibling leaf nodes. If the scan size is set to be too small, the GPU will wait until the CPU finds the starting point of the next leaf node scan (`scan_start`). If the scan size is chosen to be too large, the CPU will wait for the GPU to finish its brute-force leaf node scanning. This scanning will ultimately access a large number of data points that do not overlap a given query range.

The detailed co-processing algorithm is described in Algorithm 1. First, the upper tree is traversed by the CPU. `TraverseUpperTreeOnCPU()` is a function that the CPU executes to traverse the upper tree in host memory(line 6). Note that this function visits the child nodes from left to right when multiple child nodes overlap a given query. Therefore, the starting leaf node index of leaf node scanning `scan_start` monotonically increases from left to right. If `TraverseUpperTreeOnCPU()` finds that there are no more overlapping nodes in the upper tree, it returns 0 and the query processing will be terminated (line 8∼10). If `start_scan` is not zero, we execute the kernel function - `ScanLeafArrayOnGPU()` for brute-force scanning(line 13). Since the main CPU thread does not wait for GPU function to finish, `TraverseUpperTreeOnCPU()` and `ScanLeafArrayOnGPU()` are executed asynchronously. That is, CPU traverses the upper tree and identifies which parts of the leaf array needs to be scanned while GPU scans previously identified portion of the leaf array.

Instead of scanning multiple sibling leaf nodes, we may consider simply increasing the size of a leaf node. However, as demonstrated in our experiments, the `scan_size` needs to be adjusted per query because the optimal `scan_size` varies depending on the range query selection ratio. The selection ratio is the portion of data points that overlap a given range query. In other words, if all leaf nodes overlap a given query, the selection ratio is 1. If none of the leaf nodes overlap, the selection ratio is 0. As the selection ratio increases, a larger number of leaf nodes must be visited.

Figure 1 illustrates an example of co-processing a multi-dimensional range query with Hybrid tree index. First, we set *scan_end* to 0 and search the upper tree from root node (①). In the root node, we compare the MBB of the first child node ($R1$) with a given query. Suppose $R1$ overlaps the query and its leaf node index (6) is greater than `scan_end` (0), we stop further comparisons and visit the child node $I1$ (②). In node $I1$, we compare the query with the MBB of each child node from left to right. Assume that $R3$ does not overlap the query; however, $R4$ and $R5$ overlap. Because $R4$'s leaf node index (4) is greater than `scan_end` (0), we visit its child node $L2$. In $L2$, we find that $R13$ overlaps the query range (③) and launch the GPU kernel function to scan the leaf array on the GPU (④). Although $R13$ is the MBB of only three data points (13, 14, and

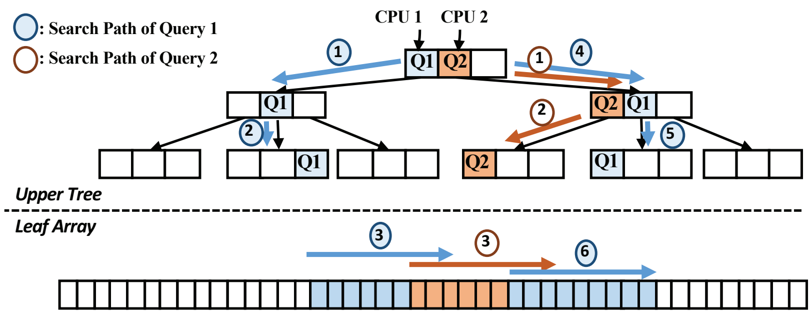


Figure 2: Multiple Query Scheduling

15 in this example), the GPU kernel function compares one leaf node with the query and with a larger number of leaf nodes specified by the `scan_size` (④). In this example, the GPU kernel function scans five leaf nodes (total 15 data points in total). After launching the GPU kernel function to process the leaf array, we immediately start the tree traversal for the same query but with an updated `scan_end` parameter. Because the GPU kernel function will consider the data points up to 27, the next tree traversal only looks for the internal tree nodes whose leaf node index is greater than 8 (27 data points). In the root node (⑤), we compare the query with the first entry. Although $R1$ overlaps the query range, its leaf node index (6) is smaller than `scan_end` (8). Hence, this time we do not visit $I1$. Assume that the next MBB $R2$ overlaps the query range. Because $R2$'s leaf node index (14) is greater than `scan_end` (8), we visit $I2$. In $I2$ (⑥), we find that the first MBB $R6$ overlaps the query range. However, because its leaf node index (8) is not greater than `scan_end` (8), we do not visit $L4$ but $L6$ (⑦). In $L6$, we launch the GPU kernel function with new `scan_start` (12) (⑧). After launching the GPU kernel function, we update the `scan_end`. When we return to the root node of the upper tree for the next traversal, we find that the largest leaf node index is not greater than the `scan_end` and finish the query processing.

3.3. Multiple Query Scheduling

When multiple queries arrive in a batch, our co-processing scheme spawns multiple CPU threads and assigns a query to each thread in order to concurrently navigate the internal tree nodes. The search paths of multiple queries are commonly different and each query needs to scan different parts of the leaf array as shown in Figure 2. When a thread reaches the parent node of a leaf node (step ② in Figure 2), we determine the scan size and the number of GPU blocks to be used for scanning the leaf array. If there is a single outstanding query, we can use all available GPU blocks for it. However, when multiple queries are concurrently executed, a smaller number of GPU blocks should be allocated to each query.

Although the multiple query scheduling problem has proved to be an NP-complete problem [23], it has also been proved that multiple heuristic approaches effectively reduce the query processing time and improve the query processing throughput. For heterogeneous range query co-processing, we propose a heuristic GPU block scheduling algorithm that adjusts the number of GPU blocks based on each query's selection ratio.

Queries that have a high selection ratio perform better when we increase the scan size and the GPU accesses more leaf nodes in parallel. Therefore, we need to assign more GPU blocks to them so that they can scan a larger number of leaf nodes with higher parallelism. However, it is not an easy problem to know the selection ratio of a query in advance. This is because the selection ratio depends on the distribution of the datasets. Hence, a larger query range does not always result in a higher selection ratio. In our heuristic GPU block scheduling algorithm, we predict the selection ratio of a query as follows.

After determining the leftmost leaf node that has an overlapping MBB, we scan the MBBs in the current node (the parent of the leaf node) from right to left to find the rightmost overlapping leaf node. If the rightmost overlapping leaf node is far from the leftmost one, it is likely to have high selection ratio. Thus, we assign more GPU blocks to the query. Otherwise we assign a small number of GPU blocks. In other words, if the offset distance between the leftmost and rightmost leaf nodes is greater than k, we assign k GPU blocks to the query. If the offset distance is smaller than 8, we use 4 GPU blocks in our implementation.

4. Evaluation

We now evaluate and analyze the performance of heterogeneous query co-processing using Hybrid tree. For the upper tree in the host memory, we implemented two versions: an R-tree and Linear Bounding Volume Hierarchy (LBVH) [24]. LBVH is a linear BVH that employs Morton codes to reduce the size of the minimum bounding box. Because we construct the upper tree in a bottom up fashion, we observe that the search performance and construction time of LBVH are slightly faster than those of the R-tree. Hence, we present the performance of Hybrid tree that employs LBVH as its upper tree. For the parallel scanning of a leaf node array, we implemented a GPU kernel function using CUDA 7.0, which scans the SoA array in a brute force fashion. We compare the performance of Hybrid tree with that of an MPHR-tree [9], which is the state-of-the-art multi-dimensional index on the GPU. We also compare its performance with that of a legacy R-tree [1] that executes only on the CPU and a variant of the R-tree called *R-tree(LeafOnGPU)*, which we modified for heterogeneous co-processing. In R-tree(LeafOnGPU), we set the size of leaf nodes to be 512x larger than that of normal tree nodes, which corresponds to a `scan_size` of `512` in Hybrid tree. As in our Hybrid tree, R-tree(LeafOnGPU) co-processes range queries, i.e., the CPU traverses internal tree nodes. Once it reaches a leaf node, the GPU scans the large leaf node in parallel, which contains as many as 98,304 data points. R-tree(LeafOnGPU) also asynchronously calls the GPU kernel function. Hence, while the GPU scans the data points in the leaf node, the CPU returns to its parent node and continues to traverse internal tree nodes until it finds another leaf node that overlaps with the given range query.

4.1. Experimental Environment

We conduct experiments on an Ubuntu 14.10 Linux machine that is powered by dual 8 core Intel Xeon E5-4620 (2.0 GHz) GPUs with hyper-threading enabled, 128 GB DDR3 memory, and dual NVIDIA Tesla K20m GPUs. The Tesla K20m GPU has 13 streaming multiprocessors (SM) and 5 GB global memory.

In our experiments, we use real datasets - the three-dimensional Integrated Surface Database (ISD) point datasets - which are available for download from NOAA' National Climatic Data Center. The datasets are associated with two-dimensional geographic information (latitude and longitude coordinates) and time as well as numerous sensor values collected from over 20,000 stations such as wind speed and direction, temperature, pressure, precipitation, etc. For the experiments, we index 40 million points, each comprising latitude, longitude, time, and a pointer to the sensor values. The datasets were collected from 2010 to 2012. In addition to the real datasets, we also synthetically generated 100 million point datasets with various distributions; however, we do not present these experiments except for those with the uniform distribution because the results of the other distributions are similar. For the real and synthetic datasets, we synthetically generated five sets of queries with various selection ratios (0.01%, 0.05%, 0.25%, 1.25%, and 6.25%).

4.2. GPU Kernel Launch Overhead

In the first set of experiments, we evaluate the kernel launch overhead and the effect of the clustering property of the leaf node array. In Table 1, we index three-dimensional 40 million data points of NOAA Integrated Surface Database (ISD) datasets. We set the number of GPU blocks and `scan_size` to 128, 256, respectively. Then, we assign one GPU block for two leaf nodes, which spawns 192 GPU threads that process two data points per thread (49 K points in total). When the data points in the leaf node array are not clustered, a single query with a selection ratio of 0.01% traverses the internal tree nodes approximately 39 times on average. When we cluster the data points in the leaf array using a Hilbert curve, the number of internal tree node traversals decreases to 2.54

Table 1: GPU Kernel Launch Overhead with or without Hilbert Space Filling Curve and K-means Clustering

	Unclustered	Hilbert Curve	K-Means
Time(msec)	0.81	0.046	0.065
GPU Kernel Launches	39.60	2.54	3.27
CPU Node Visits	165.29	14.77	19.07
GPU Node Visits	10136.67	650.03	837.57

(1/15th of 39) and the query response time decreases by 1/17. We also clustered the data points using a k-means clustering algorithm. We varied the number of clusters (k) as the k-means clustering algorithm does not group an equal number of points per cluster. In other words, even if k is set to the number of leaf nodes, a single cluster can span multiple leaf nodes, which can result in a larger number of overlaps among the MBBs than for the MBBs generated from the Hilbert curve. Compared to the unclustered leaf node array case, k-means clustering requires fewer GPU kernel launches and reduces the query response time. However, we find that k-means clustering falls short and it is outperformed by Hilbert curve clustering.

4.3. Various GPU Thread Block & Scan Size

In the experiments shown in Figure 3 and Figure 4, we index 100 million three-dimensional data points and measure the amount of memory accessed in accordance with varying `scan_size` and the number of GPU blocks and the average query response time of 1,000 queries. When the scan size is set to a single leaf node (i.e., the number of GPU blocks is equal to the `scan_size`), its performance is mostly determined by the performance of searching the upper tree using the CPU, except it has an additional overhead of launching GPU

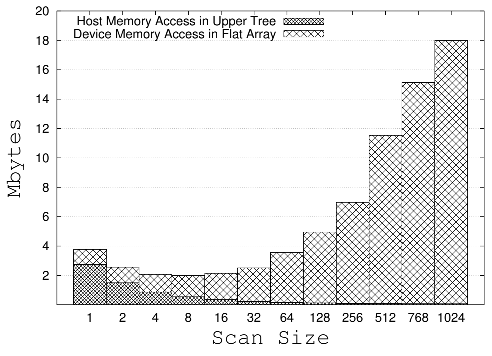


Figure 3: Amounts of Host and Device Memory Access with Varying Scan Size

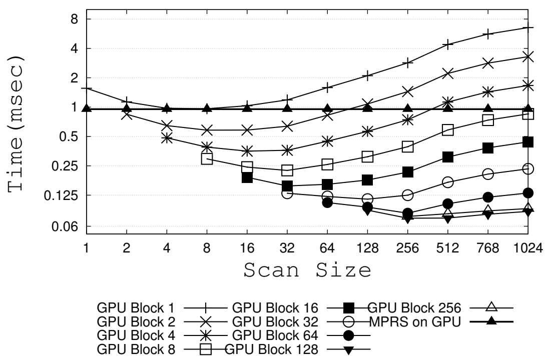


Figure 4: Query Execution Time of Various GPU Thread Blocks and Scan Size

kernel function for each leaf node. As shown in Figure 3, most of the accesses to the index occur in the upper tree. However, as we increase the scan size, more workload is assigned to the GPU kernel function and the CPU accesses fewer internal tree nodes.

We can adjust the amount of GPU workload using two parameters; the number of GPU blocks and the number points processed by a single GPU thread. When we fix the number of blocks and increase the number of points per thread, the query response time becomes faster up to a certain extent. If we assign only one data point per GPU thread, we observe that the overhead of creating a GPU thread becomes the dominant performance factor and it affects the query response time. However, if we make a single GPU thread process too many

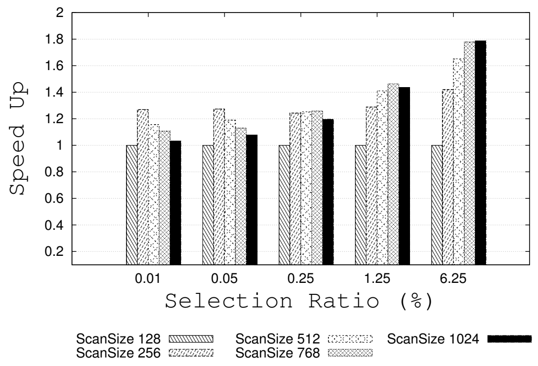


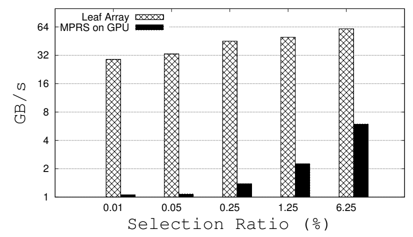
Figure 5: Speed Up with Various Selection Ratio and Scan Size(Normalized to 128)

points, the GPU visits too many non overlapping data points, which also affects the query response time. As shown in Figure 3, a larger scan size causes more GPU device memory to be accessed.

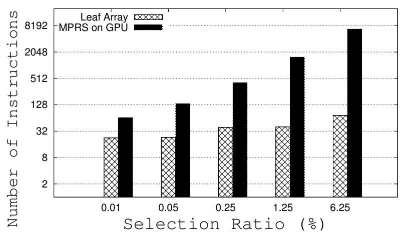
Note that the GPU block creation overhead can be different across the GPU architecture models and platforms. However, in our testbed machine - K20m, processing four data points per each GPU thread results in the best performance in most cases. In the worst case, the query response time becomes 6x slower when each thread processes 1,024 data points. These results show that choosing the right workload per GPU thread is a key performance factor in query co-processing.

When we increase the number of GPU blocks instead of increasing the number of data points per thread, it better utilizes a large number of streaming multiprocessors in the GPU and the query processing performance improves. Note that multiple thread blocks concurrently access different parts of the leaf array. However, if the number of GPU blocks exceeds the number of available GPU blocks (208 in K20m), the performance improvement saturates as the extra GPU blocks are serialized.

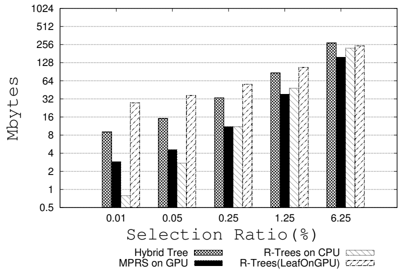
When we use 128 GPU blocks and each thread processes two data points, the query response time is 12x faster than that of the MPRS algorithm with the MPHR-tree. Note that the amount of memory access is minimized when the



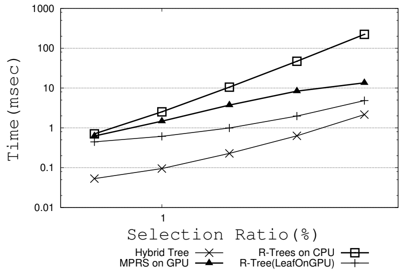
(a) *Global Memory Load Throughput per Second*



(b) *Average Number of Executed Conditional Branch Instructions per Query*



(c) *Amounts of Memory Access with Varying Selection Ratio per Query*



(d) *Average Query Execution Time with Various Selection Ratio per Query*

Figure 6: Performance Results with Various Selection Ratio

scan size is 8 leaf nodes; however, its query response time is not smaller than that when the scan size is 256 leaf nodes. This is because even though we access a larger amount of GPU device memory, the memory is concurrently accessed and it does not degrade the performance.

Figure 4 also shows the query response time of the MPRS tree traversal algorithm using the MPHR-tree. The MPRS algorithm is similar to our heterogeneous parallel index co-processing in the sense that it also performs brute-force linear scanning for leaf nodes. However, the MPHR-tree manages all internal nodes in the GPU device memory and it navigates the internal tree nodes in the GPU. The MPRS algorithm and MPHR-tree are comprehensively explained in the original MPHR-tree paper [9]. Although the MPRS algorithm eliminates backtracking and accesses mostly contiguous memory blocks, it irregularly visits internal tree nodes because the branch prediction of the hierarchical indexing structures is difficult. The performance improvement of our heterogeneous co-processing over the MPHR-tree mostly comes from the overhead of the internal tree node traversals.

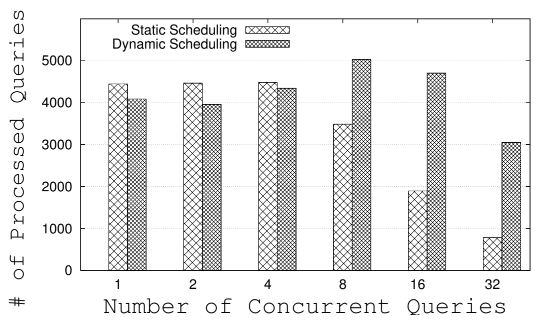
In the experiments shown in Figure 5, we set the number of GPU blocks to 128 and varied the scan size. When a GPU block processes a single leaf node, it suffers from the high kernel launch overhead and shows the worst performance. When the selection ratio is higher than 0.25%, a larger `scan_size` effectively reduces the number of internal tree node traversals and leaf node array scans. However, making each thread process more leaf nodes does not always result in better performance. When the selection ratio is lower than 0.25%, a large `scan_size` performs poorly because it increases the number of unnecessarily accessed data points that do not overlap.

Figure 6(a) shows the global memory load throughput that we measured using *nvprof* profiler. In the experiments, we varied the selection ratio of range queries. The global memory load throughput shows the amount of bytes loaded from the GPU device memory per second. In the Tesla K20m GPU, the theoretical max bandwidth of device memory is 200 GB/s. While the device memory load throughput of our Hybrid tree is 28 $\sim$ 61 GB/s, that of the MPHR-tree is less than 10 GB/s. This demonstrates that the MPRS algorithm fails to leverage the high bandwidth of GPU global memory because of its irregular internal tree node traversals.

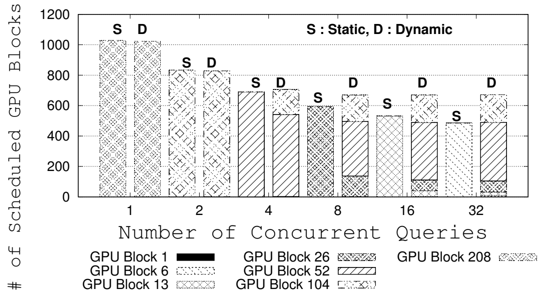
Figure 6(b) shows the number of average conditional branch instructions executed per query. As we increase the selection ratio, more tree nodes are accessed and more `if` statements are called. When the selection ratio is 6.25%, the MPRS algorithm requires a 94x number of conditional branches, which degrades the global memory load throughput and query response time.

Figure 6(c) shows the average number of bytes accessed per query. When the selection ratio is 0.01%, Hybrid tree accesses approximately 3x the amount of bytes than MPRS. However, when the selection ratio is 6.25%, Hybrid tree accesses no more than 2x the number of bytes from the device memory. Note that the global memory load throughput of Hybrid tree is more than 10x higher than that of MPRS when the selection ratio is 6.25% as shown in Figure 6(a). This indicates that although MPRS accesses a similar amount of device memory, its internal tree node traversal and branch divergence problem prevent it from taking advantage of the high memory bandwidth. When the selection ratio is 0.01%, the legacy R-tree accesses only 0.7 MB of the host memory. However, as the selection ratio increases, it accesses almost similar amounts of memory as other co-processing schemes. As for R-tree(LeafOnGPU), it has a very large leaf node; however, its node utilization is not 100%, unlike the MPHR-tree and Hybrid tree. This is because we construct the R-tree structure in a top-down manner without using space-filling curve. For these reasons, R-tree(LeafOnGPU) accesses a much larger number of bytes than the others when the selection ratio is small.

Figure 6(d) shows the query response time. For Hybrid tree, we use 128 GPU blocks and we set the scan size to 512. That is, each GPU block scans four leaf nodes. When the selection ratio is 0.01%, the query response time of Hybrid tree is 8x, 12x, and 14x faster than that of R-tree(LeafOnGPU), MPRS, and R-tree respectively. Although MPRS uses the GPU, its performance with a very low selection ratio is not significantly better than that of the legacy R-tree on the CPU because MPRS spends most of its execution time on internal node visits and spends less time on brute-force linear scanning of the leaf nodes. R-tree(LeafOnGPU) outperforms MPRS and the legacy R-tree by leveraging both the CPU and GPU. However, it suffers from poor node and CPU utilization. Because the size of leaf nodes is considerably large, the number of internal tree nodes is much smaller than that of Hybrid tree. Hence, the CPU often becomes idle while the GPU is processing a large leaf node. Decreasing the size of the leaf nodes degrades the performance because the GPU kernel function get launched more frequently. For these reasons, R-tree(LeafOnGPU) is consistently outperformed by Hybrid tree. When the selection ratio is 6.25%, Hybrid tree is 2x and 6x faster than R-tree(LeafOnGPU) and MPRS respectively.



(a) *Query Throughput per Second*



(b) *Number of Scheduled GPU Blocks with Varying Concurrent Queries*

Figure 7: Static vs. Dynamic Query Scheduling

4.4. Throughput of Batch Query Processing

When queries arrive in a batch, our co-processing scheme spawns multiple CPU threads and schedules the GPU blocks for concurrent co-processing. Figure 7 shows the effectiveness of our dynamic GPU block scheduling algorithm. In our experiments, we varied the size of query batches and selection ratio of each query.

As a baseline, we show the performance of *static* scheduling, which assigns $208/N$ GPU blocks to each query, where N is the number of queries. In other words, when there is only one query submitted, we assign 208 GPU blocks to the query. If there are 4 concurrent queries, we process each query using 52 GPU blocks.

In our dynamic GPU block scheduling algorithm, we determine the number of GPU blocks according to the distribution of overlapping MBBs in the parent of the leaf node, as described in section 3.3. If the offset distance between overlapping MBBs is greater than 104, we assign 104 GPU blocks to each query. Hence, if more than 2 queries need 104 GPU blocks, the number of scheduled GPU blocks will exceed the capacity of our testbed K20m GPU.

Figure 7(b) shows the number of GPU blocks scheduled per query when the number of concurrent queries is varied. As each batch comprises multiple queries, the total number of launched GPU blocks decreases when static scheduling is employed; however, it does not decrease linearly. This is because with a smaller scan size, we need more internal node traversals, which call more GPU kernel functions.

When a batch comprises a small number of queries, our dynamic scheduling obtains a performance similar to that of static scheduling. However, as the batch contains more than 4 queries, our dynamic GPU block scheduling algorithm effectively adjusts the scan size for each query and utilizes the GPU blocks more efficiently than static scheduling. Therefore, when the number of concurrent queries is between 4 and 16, the query processing throughput is higher than that in a case wherein a single query is processed at a time (i.e., the number of concurrent queries = 1). Compared to static scheduling, our dynamic scheme yields approximately 1.5x, 2.5x, and 4x higher query processing throughput when the batch size is 8, 16, and 32 respectively.

5. Conclusion

In this work, we presented a novel multi-dimensional range query co-processing scheme that utilizes both the CPU and GPU. Because the large branching factor in a tree-structured index makes it difficult to parallelize tree traversal algorithms, we make use of the CPU for hierarchical tree traversal and the GPU for brute-force linear scanning. In the co-processing scheme, balancing the workload between the CPU and GPU is important for improving the performance.

To leverage both the CPU and GPU, we asynchronously call the GPU kernel function that scans multiple leaf nodes in parallel and restart the internal node traversal while the GPU is accessing leaf nodes. This co-processing scheme effectively overlaps the CPU and GPU computations in time. Additionally, our co-processing scheme considers the selection ratio of range queries to adjust the workload between the CPU and GPU.

We believe that this is the first work that proposes a GPU block scheduling algorithm for multiple multi-dimensional range queries. Our proposed scheduling algorithm determines the number of GPU blocks to be used based on the selection ratio of each query predicted while traversing the internal tree nodes. The key idea of our multiple query scheduling algorithm is to assign more GPU blocks to the queries that can be rapidly processed by sequentially accessing a large number of leaf nodes.

Our performance study using the real and synthetic datasets confirms that the proposed heterogeneous co-processing scheme improves the query response time by up to 12x and the query processing throughput by up to 4x.

6. acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016M3C4A7952634 and 2.160233.01)

References

[1] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proceedings of 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD), 1984.

[2] A. Eklund, P. Dufort, D. Forsberg, S. M. LaConte, Medical image processing on the GPU – past, present and future, Medical Image Analysis 17 (8) (2013) 1073 – 1094.

[3] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (16) (2007) 2618–2640.

[4] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, et al., Radiative signatures of the relativistic kelvin-helmholtz instability, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 5.

[5] B. Jang, D. Schaa, P. Mistry, D. Kaeli, Exploiting memory access patterns to improve memory performance in data-parallel architectures, IEEE Transactions on Parallel and Distributed Systems 22 (1) (2011) 105–118.

[6] V. Garcia, E. Debreuve, M. Barlaud, Fast k nearest neighbor search using gpu, in: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE, 2008, pp. 1–6.

[7] V. Garcia, E. Debreuve, F. Nielsen, M. Barlaud, K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching, in: 2010 IEEE International Conference on Image Processing, IEEE, 2010, pp. 3757–3760.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. w. Lee, S. A. Brandt, P. Dubey, FAST: Fast architecture sensitive tree search on modern CPUs and GPUs, in: Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2010.

[9] J. Kim, W.-K. Jeong, B. Nam, Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu, Transactions on Parallel and Distributed Systems 26 (8) (2015) 2258–2271.

[10] A. Shahvarani, H.-A. Jacobsen, A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms, in: Proceedings

of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2016.

[11] J. Kim, S. Hong, B. Nam, A performance study of traversing spatial indexing structures in parallel on GPU, in: 3rd International Workshop on Frontier of GPU Computing (in conjunction with HPCC), 2012.

[12] J. Kim, B. Nam, Parallel multi-dimensional range query processing with r-trees on GPU, Journal of Parallel and Distributed Computing 73 (8) (2013) 1195–1207.

[13] D. Foley, Nvlink, pascal and stacked memory: Feeding the appetite for big data, Nvidia. com.

[14] T. Foley, J. Sugerman, KD-tree acceleration structures for a gpu raytracer, in: ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2005.

[15] M. Hapala, T. Davidoiv, I. Wald, V. Havran, P. Slusallek, Efficient stackless BVH traversal for ray tracing, in: the 27th Spring Conference on Computer Graphics (SCCG '11), 2011.

[16] S. Popov, J. Gunther, H.-P. Seidel, P. Slusallek, Stackless KD-tree traversal for high performance GPU ray tracing, in: Eurographics, 2007.

[17] B. Smits, Efficiency issues for ray tracing, Journal of Graphics Tools 3 (2) (1998) 1–14.

[18] D. Horn, J. Sugerman, M. Houston, P. Hanrahan, Interactive k-d tree GPU raytracing, in: Symposium on Interactive 3D Graphics and Games (I3D), 2007.

[19] L. Luo, M. D. Wong, L. Leong, Parallel implementation of r-trees on the GPU, in: 17th Asia and South Pacific Design Automation Conference (ASP-DAC), 2012.

[20] B. Moon, H. V. Jagadish, C. Faloutsos, J. H. Saltz, Analysis of the clustering properties of the hilbert space-filling curve, IEEE Transactions on Knowledge and Data Engineering 13 (1) (2001) 124–141.

[21] N. Govindaraju, J. Gray, R. Kumar, D. Manocha, GPUTeraSort: high performance graphics co-processor sorting for large database management, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 325–336.

[22] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for gpu computing, in: Graphics hardware, Vol. 2007, 2007, pp. 97–106.

[23] B. Nam, M. Shin, H. Andrade, A. Sussman, Multiple query scheduling for distributed semantic caches, Journal of Parallel and Distributed Computing 70 (5) (2010) 598–611.

[24] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast bvh construction on gpus, in: Computer Graphics Forum, Vol. 28, Wiley Online Library, 2009, pp. 375–384.