Exploiting Massive Parallelism for Indexing Multi-dimensional Datasets on the GPU

Jinwoong Kim, Won-Ki Jeong, *Member, IEEE,* and Beomseok Nam, *Member, IEEE*

Abstract—Inherently multi-dimensional n-ary indexing structures such as R-trees are not well suited for the GPU because of their irregular memory access patterns and recursive back-tracking function calls. It has been known that traversing hierarchical tree structures in an irregular manner makes it difficult to exploit parallelism and to maximize the utilization of GPU processing units. Moreover, the recursive tree search algorithms often fail with large indexes because of the GPU's tiny runtime stack size. In this paper, we propose a novel parallel tree traversal algorithm - *Massively Parallel Restart Scanning (MPRS)* for multi-dimensional range queries that avoids recursion and irregular memory access. The proposed MPRS algorithm traverses hierarchical tree structures with mostly contiguous memory access patterns without recursion, which offers more chances to optimize the parallel SIMD algorithm. We implemented the proposed MPRS range query processing algorithm on n-ary bounding volume hierarchies including R-trees and evaluated its performance using real scientific datasets on an NVIDIA Tesla M2090 GPU. Our experiments show braided parallel SIMD friendly MPRS range query algorithm achieves at least 80% warp execution efficiency while task parallel tree traversal algorithm shows only 9%-15% efficiency. Moreover, braided parallel MPRS algorithm accesses 7∼20 times less amount of global memory than task parallel parent link algorithm by virtue of minimal warp divergence.

Index Terms—Parallel multi-dimensional indexing; Multi-dimensional range query; GPGPU;

1 Introduction

General purpose computing on graphics processing units (GP-GPU) is now widely used for high performance parallel computation as a cost effective solution [24]. GPUs enable large independent datasets to be processed in a SIMD (single instruction multiple data) fashion, so a broad range of computationally expensive but inherently parallel computing problems, such as medical image processing [4], scientific computing [14], and computational chemistry [31], have been successfully accelerated by GPUs. While GPU-accelerated systems achieve superior performance for computationally expensive scientific applications that involve matrix manipulations, there still exist many scientific computing domains that have not yet leveraged the parallel computing power of the GPU.

In many scientific disciplines, datasets are commonly made up of collections of multi-dimensional arrays where each array element has spatial and temporal coordinates; for example, location and time information from sensor devices in two or three dimensional spaces. In order to help navigate through such multi-dimensional scientific datasets, many scientific data file formats, such as NetCDF [26] and Hierarchical Data Format(HDF) [8], have been developed. However, such data formats still do not fully support multi-dimensional indexing that allows direct access to the subsets of datasets using ranges of spatial and temporal coordinates. Since one of the most common access patterns for such scientific datasets is multi-dimensional range query, some external indexing libraries such as GMIL and FastQuery have been developed to improve the range query performance [22], [3].

The multi-dimensional indexing tree structures are used not only for high performance scientific data analysis applications, but also for general purpose applications such as geographic information systems, collision detection in computer graphics, etc. In the computer graphics community, bounding volume hierarchy (BVH) tree structures and kd-trees are the most commonly used data structures for ray tracing and collision detection [29], [5]. The database community also improved multi-dimensional indexing structures for the past couple of decades. R-Tree [9] and its variants are most commonly used for multi-dimensional range query processing as of today. R-Tree can be considered as a particular class of BVH as it uses hierarchically wrapping bounding boxes but has a degree larger than 2.

Regardless of their popularity, hierarchical multi-dimensional indexing trees are known to be inherently not well-suited for parallel processing due to their irregular and recursive back-tracking tree traversal patterns [19]. In computer graphics, task parallelism is exploited to utilize a large number of GPU cores, i.e, each processing unit in GPU traverses a binary BVH for different rays or objects. With such task parallelism approach, a very large number of queries can be processed concurrently, but it does not improve the response time of each individual query. As another form of parallelism, data parallelism can be exploited to make each individual query run faster. However, traversing a hierarchical tree structure in SIMD fashion for a multi-dimensional range query can make it difficult to reason about load balancing. Moreover, range query processing needs to access trees in an irregular and recursive manner, which makes it even more

• *The authors are with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology,Ulsan, 689-798, Republic of Korea.*
E-mail: {jwkim, wkjeong, bsnam}@unist.ac.kr

difficult to exploit data parallelism and to maximize the utilization of the GPU cores. In addition, due to tiny shared memory size used as the run-time stack on the GPU, the recursive tree traversal for a large index size may cause a stack overflow and fail to work. The traditional multi-dimensional indexing tree search algorithms need back-tracking to previously visited tree nodes since multi-dimensional range queries or nearest neighbor queries may require to visit multiple children of a tree node. Therefore, the legacy R-tree and BVH search algorithms make the previously visited tree nodes reside in the run-time stack while navigating their sub-trees. In order to resolve the tiny run-time stack problem, several tree traversal methods have been proposed in the computer graphics community, such as *kd-restart* [7], *fixed short stack* [12], *rope tree* [30], and *parent link* [10] search algorithms. However, these stackless tree traversal algorithms were studied only for task parallelism, and were never considered for data parallel n-ary trees such as R-trees.

In this work, we resolve the tiny run-time stack problem for n-ary tree structures by proposing a novel stackless tree traversal algorithm - *Massively Parallel Restart Scanning (MPRS)*, which is designed for a novel variant of R-trees - *Massively Parallel Hilbert R-tree (MPHR-tree)*. We also extend the *short stack*, *skip pointer* and *parent link* algorithms for data parallel n-ary tree structures, and show that our novel MPRS algorithm outperforms them. The contributions of this work are summarized as follows.

- **Massively Parallel Processing of N-ary Multi-dimensional Index**
  In order to maximize the core utilization of the GPU, we let the degree of indexing tree nodes to be a multiple of the number of cores in a GPU streaming multiprocessor, so that all the bounding boxes of a single tree node can be compared against a given query in parallel while avoiding warp-divergence. *Data parallel* or *braided parallel* [6] indexing shows significantly higher performance than *task parallel* indexing in terms of global memory access reduction and SIMD efficiency.
- **Massively Parallel Hilbert R-Tree and MPRS Range Query Algorithm**
  Multi-dimensional range query may overlap multiple bounding boxes of a single tree node. Hence, legacy multi-dimensional range query algorithms use recursion or stack, and visit the overlapping child nodes in depth-first order. However, since the run-time stack on the GPU is tiny, we develop a variant of multi-dimensional indexing trees - *MPHR-tree*, where each tree node embeds the largest leaf index (monotonically increasing sequence number of a leaf node, or a Hilbert value) of its sub-tree, which helps avoid the recursion and irregular memory access. The embedded leaf index is necessary for a novel multi-pass range query algorithm - *MPRS (Massively Parallel Restart Scanning)* that traverses the MPHR-tree structures in a mostly sequential fashion. MPRS avoids visiting the already visited nodes by keeping track of the largest index of visited leaf nodes.
- **Comparative Performance Study of Skip Pointer, Short Stack, and Parent Link Algorithm for N-ary Multi-dimensional Indexing Trees**
  Skip pointer [30], short stack [12] and parent link [10] are the traversal algorithms for multi-dimensional indexing tree structures used in ray tracing to resolve the tiny run-time stack problem of the GPU. These stackless ray tracing algorithms are not designed to traverse tree structures in data parallel fashion but in task parallel fashion, i.e., each GPU processing unit processes its own individual ray. In this work, we adopt skip pointer, short stack, and parent link algorithms for n-ary multi-dimensional indexing structures, make a block of GPU threads concurrently process a single query, and compare their performance against our MPRS range query algorithm.

In our experiments, we show braided parallel MPRS algorithm yields higher than 80% warp execution efficiency which is an indication of SIMD efficiency, but task parallel indexing shows less than 20% efficiency. Also braided parallel MPRS algorithm accesses up to 20 times less amount of data from global memory than task parallel stackless tree traversal algorithms. As a result, braided parallel MPRS algorithm shows more than 3 times faster average query response time than task parallel tree traversal algorithms. As for the query processing throughput, braided parallel MPRS range query processing algorithm processes more than 56,000 queries per second using a single Tesla M2090 GPU, which is 364 times higher throughput than multi-threaded R-trees on AMD 8 core Opteron 6127 CPUs.

The rest of the paper is organized as follows. In Section 2 we discuss previous literature related to GPU-based tree traversal algorithms. Section 3 discusses R-Trees and its implementation on the GPU and our adaptation of stackless tree traversal algorithms for multi-dimensional query processing. In Section 4, we propose a novel MPHR-tree indexing structure and MPRS range query processing algorithm that avoids backtracking when traversing the indexing trees. We discuss how its parallel construction and search operation can be performed in a SIMD fashion. In Section 5, we show performance improvements provided by MPHR-trees in comparison with other stackless traversal algorithms. Section 6 concludes the paper.

2 Related Work

NVIDIA has been increasing the number of concurrent threads per streaming multiprocessor in their product line-up of GPUs, but the shared memory sizes of them have been minimally enlarged [1]. CUDA threads use the fast shared memory as their run-time stack, but the latest Tesla GPUs have only 48K bytes of shared memory. Due to their tiny stack sizes, the recursive search algorithms of multi-dimensional indexing structures often fail. Although a large number of multi-dimensional indexing structures have been proposed, the search algorithms of those methods are similar in a sense that they recursively prune out the sub-trees depending on whether a given query range overlaps the bounding boxes of sub-trees.

In the computer graphics community, there have been a large amount of efforts to handle the small stack problems on the GPU. Foley et al. [7] proposed *kd-restart* algorithm for ray tracing using kd-trees. The kd-restart algorithm can not be directly applied to multi-dimensional range query processing since it tracks crossing points of the ray with the region boundaries of kd-tree leaf nodes. Later, Hapala et al. [10] proposed a traversal algorithm - *parent link* for bounding volume hierarchies that does not need a stack and minimizes the memory needed for ray tracing. In their proposed method, each tree node has a pointer to its parent node so that when it can back-track to a parent node by following the pointer. Horn et al. [12] extended Foley's kd-restart algorithm by employing a small fixed-sized stack so that the number of restart from the root node is minimized. Although kd-restart algorithm can not be employed for multi-dimensional range query processing, the stackless tree traversal algorithms proposed by Hapala et al. [10], Horn et al. [12], and Smits et al. [30] can be extended to n-ary tree structures for parallel query processing by simple modification of the algorithms, which will be discussed in detail in Section 3.2.

Foley, Hapala, Smits, and Horn's algorithms do not leverage multiple GPU cores for a single ray, but a group of rays are processed in parallel on the GPU. In order to utilize multiple cores of GPU streaming multiprocessor (SM), Kim et al. proposed *FAST (Fast Architecture Sensitive Tree)*, which rearranges a binary search tree into tree-structured blocks to maximize data-level and thread-level parallelism on the GPU [17]. Each block of FAST is the unit of parallel processing in a single SM, which is similar to the tree node of n-ary trees such as B-Tree, but the block size of FAST is chosen to avoid the bandwidth bottleneck between main memory and GPU device memory. The difference between our work and FAST is that FAST is designed for one-dimensional range or exact match query, therefore it does not need back-tracking, i.e., query processing is completed after a leaf node reached.

Although GPUs have a large number of processing units, each processing unit of the GPU runs much slower than a CPU core. Hence, several researches have been conducted to improve both response time and query processing throughput by assigning an independent query to each streaming multiprocessor on the GPU and make multiple cores of SM process the same query, which is called *braided parallelism* [6]. Fix et al. proposed a braided parallel one-dimensional query processing method for B+-trees, wherein a single B+-tree exists in global memory of GPU and multiple independent queries are concurrently processed across SMs, and a block of threads in each SM process individual query in parallel [6]. Zhou et al. also implemented parallel B+-tree that compares multiple keys at the same time using SIMD instructions [32]. Kaldewey et al. [15] proposed a parallel index called *P-ary search*, for one dimensional sorted lists.

In order to exploit multiple cores of the GPU and to let them process a single multi-dimensional range query in parallel, Luo et al. [20] proposed a parallel R-Tree traversal algorithm on the GPU. However their method employs a small queue in the shared memory of streaming multiprocessor in order to store the bounding box overlap information, which is not the best strategy for a tiny shared memory size. It is also not very scalable since the shared queue requires atomic write operation that significantly impairs concurrency and performance.

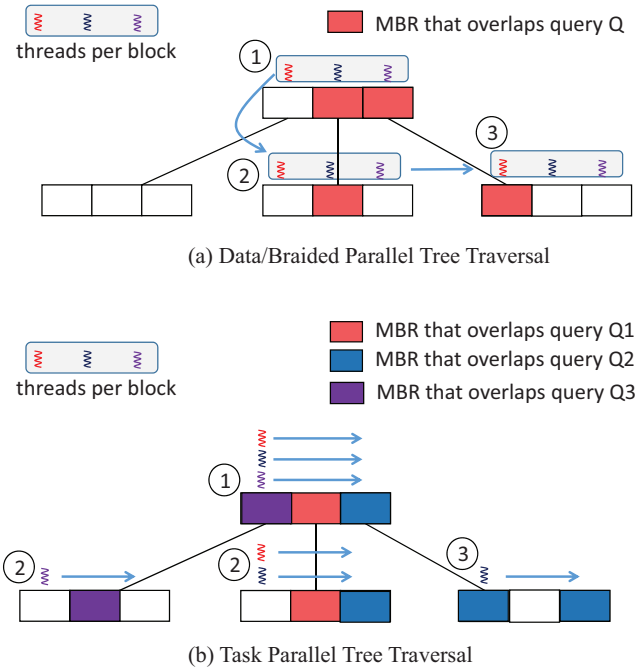


Fig. 1. *In data parallel (or braided parallel) tree traversal algorithms, multiple threads cooperate to process a single query while task parallel tree traversal algorithms let each thread access different part of tree nodes and cause warp divergence.*

3 Parallel Multi-Dimensional Indexing on the GPU

In this section, we overview parallel multi-dimensional indexing structures, and our adaptation of stackless tree traversal algorithms for multi-dimensional range query problems, which will be compared with our MPRS algorithm.

3.1 Data Parallel Multi-Dimensional Indexing

Multi-dimensional indexing trees present several challenges in parallel computation. First, tree nodes are connected using pointers. If shared memory is not provided as in shared-nothing environment, tree nodes are local only to one process. Second, the irregular tree structures and the hierarchical but irregular tree traversal patterns make it difficult to achieve good load balance and to maximize the utilization of parallel computing resources. As GPUs provide both shared and global memory, we focus on the latter challenge.

3.1.1 R-trees and their implementation on the GPU

Recently, researchers have investigated on how to utilize multiple processing units of GPU to improve the search performance in tree structured index. Zhou et al. [32] and Kim et al. [19] developed tree structured indices that allow multiple consecutive key values to be compared simultaneously by

concurrent threads. Their indexing structures are similar to that of a B-tree. Each processing unit of GPU compares the same query with its own child branch of a tree node in parallel. Searching the traditional binary tree structures requires processing of a single tree node at a time, but these novel parallel multi-dimensional indexing structures allow SIMD comparison and a large number of parallel threads process a single query to improve the search performance in terms of query response time. This is one of the major differences from tree-traversal acceleration algorithms used in computer graphics, such as kd-trees or BVH for ray tracing that leverage *task-parallelism* of the GPU, rather than *data-parallelism*. Figure 1 illustrates how data parallel traversal algorithms and task parallel traversal algorithms differ in accessing tree nodes. In data parallel tree traversal algorithms, a set of threads in a block concurrently access the same tree node but threads in task parallel tree traversal algorithms independently access different parts of tree structure. With the task parallelism, there can be varying number of tree node accesses per thread depending on the query range, thus the query execution times of queries are not homogeneous and the query turn-around times are determined by the slowest query.

Scientific instruments, such as sensors on Earth orbiting satellites and high-resolution microscopes, produce hundreds of gigabytes of spatio-temporal multi-dimensional datasets daily. In order to access such large multi-dimensional datasets efficiently, extensive research has been carried out on multidimensional indexing structures, including R-trees [9]. R-trees group nearby spatial objects, store them in a tree node, build their *minimum bounding rectangle (MBR)* for the tree node, and hierarchically store the MBR in its parent tree node. R-tree was originally designed for disk-based database systems, but it has many strengths in parallel indexing on the GPU. First, R-tree nodes can have a large number of child nodes (B), hence multiple processing units of the GPU can concurrently compare bounding boxes of child nodes for a given range query. Moreover, R-tree is a balanced tree structure, thus it bounds the height of a tree structure by $log_B N$. At each level, a block of threads access an array of MBRs that are stored in contiguous memory, thus code optimizations such as *memory coalescing*, *bank conflict avoidance*, *instruction level parallelism*, etc, can be easily implemented.

A single *warp* – the minimum thread scheduling unit in CUDA architecture – can execute the same instruction in parallel to compare multiple MBRs with a given query range in parallel. Unlike most of traversal algorithms used in ray tracing literature, where each ray is traversed independently from the others, are likely to cause warp divergence, but our parallel overlap comparison performs coherent memory access and no divergence for comparison. Each thread compares a query against one of the MBRs stored in the current node, and the threads visit one of the child nodes for that MBR if overlapped with the query. This child-visiting is done *per-warp*, not *per-thread*, so we can avoid warp divergence effectively.

In database systems, the degree of n-ary tree structures is determined by the page size of disk storage. However, for GPU indexing, the degree of tree node is set to a multiple of the number of processing units in a GPU block. In our experiments, good search performance is achieved when the degree of tree nodes is four times larger than the number of cores in a GPU block, i.e., four child nodes per GPU thread. If the degree of a tree node is 256, the R-tree node size becomes greater than 10 KB, and GPUs that have 48 KB of shared memory can store maximum four tree nodes, which means searching indexing trees taller than five is not feasible. In practice, shared memory is consumed not only to stack the tree nodes to visit but also to store shared variables to coordinate CUDA threads within a block. Thus, the recursive search function on the GPU may suffer from a stack-overflow problem even for a small tree height, like three, and stack-less traversal is preferred on the GPU.

3.2 Stackless Multi-dimensional Range Query Processing

In computer graphics, stackless ray traversal algorithms have been proposed mainly because a large number of rays are traced in parallel and the overhead of using run-time stack can be very high, i.e, the size of memory space for run-time stack can be as large as the maximum stack depth times the number of rays. Hence, several stackless traversal algorithms have been proposed for efficient ray tracing on the GPU.

Some of the stackless ray tracing algorithms can not be used for multi-dimensional range query processing, but there are other stackless search algorithms that can be adopted for multi-dimensional range queries with some modifications to the algorithms. For an instance, kd-restart ray tracing algorithm divides a ray into smaller line segments and reduces the bounding boxes of the lines. Unlike line intersection query processing, multi-dimensional range query processing can not reduce the total size of bounding box without dividing multi-dimensional query range into small fragmented regions. In this section, we list a couple of stackless ray tracing algorithms and discuss the extensions of them we adopted for multi-dimensional range query processing.

3.2.1 User-defined Stack in Global Memory

A simple way of avoiding recursion is to store the activation record information in a user-defined stack using a large but slow global memory on the GPU. However, a previous study [18] shows the global memory access becomes serious bottleneck since a large number of concurrent CUDA threads should wait for an exclusive lock when they need to perform the stack operations.

3.2.2 Kd-restart for Range Query Processing

Kd-restart algorithm [7] eliminates the stack operations by restarting the search at the root of the tree. Instead of backtracking, the kd-restart algorithm traverses a tree structure multiple times from root node to a leaf node. In each leaf node it visits, the algorithm computes a crossing point of the ray with hyper-plane boundaries of the leaf node. With the piercing point along the ray, kd-restart algorithm truncates the ray and searches the kd-tree with the updated ray from root node. Since the ray is truncated per each restarted traversal, it avoids visiting already visited leaf nodes.

The kd-restart algorithm is designed to process each ray independently using a single GPU thread. In ray tracing, it is easy to track a point along a given ray that pierces a hyper-plane of a visited leaf node. However, if a query is not a line segment but a multi-dimensional region, pruning out visited regions will not create a simple rectangular region, which will complicate the next restart. Hence, kd-restart algorithm can not be directly applied in multi-dimensional range query processing. Another problem with kd-restart algorithm is that its memory access pattern is very irregular, and the number of tree node accesses for each query is very diverse, which significantly hurts SIMD efficiency.

3.2.3 Rope Tree

In order to avoid backtracking to previously visited tree nodes, auxiliary links - *ropes* between neighboring tree nodes can be added to kd-trees [11], [25]. Havran et al. [11] proposed *rope tree* where each node has pointers called *ropes* which stores the neighboring nodes in each dimension, i.e., a 3-dimensional kd-tree node has 6 ropes. Ray tracing traversal that pierces one of the faces of a tree node can simply follow the rope of the face. Rope tree does not need stack operations since only one of the faces will be pierced by a given ray. Popov et al. [25] extended the rope tree and showed it shows higher ray tracing throughput than CPU-based ray tracers.

However, the rope tree algorithm can not be simply adopted for multi-dimensional range query processing since multi-dimensional range query does not pierce a single point of faces. Moreover, n-ary bounding volume hierarchies or R-trees should have $2 \times n$ faces and ropes per dimension. If a query overlaps multiple MBRs, more than one ropes should be traversed and thereby stack operations can not be eliminated.

3.2.4 Parent Link

Another simple strategy is to traverse the hierarchical tree structure as in graph traversal. Hapala et al. [10] proposed a *parent link* search algorithm for bounding volume hierarchies. In their proposed bounding volume hierarchies, each tree node stores a pointer to its parent node. When a tree traversal needs to backtrack to its parent node, the parent node can be fetched from global memory using the parent pointer. Although parent link algorithm eliminates the stack operations, the backtracking using parent pointer requires additional global memory accesses.

Parent link algorithm can be used not only for ray tracing but also for n-ary data parallel range query processing, thus we develop parent link algorithm for n-ary R-tree and multi-way BVH to compare against our MPRS algorithm. As we will show in the experiments section, parent link algorithm shows decent performance.

3.2.5 Skip Pointer

Skip pointer [30] is similar to rope tree in a sense that each tree node has an auxiliary link to its right sibling node or a right sibling of its parent node. Figure 2 illustrates the n-ary tree structure with skip pointers. If the current tree node is not hit by a ray, skip pointer is followed instead of backtracking to its previously visited parent node. Unlike rope tree, skip pointer does not take into account the ray direction, which is known to incur performance penalty.

Since skip pointer algorithm does not consider any direction preference, we can adopt it for multi-dimensional range query in order to avoid stack operations and make the search path always visit non-visited node. If a tree node has no overlapping child node, skip pointer algorithm follows the skip pointer to visit a right sibling node or a sibling of its parent node.

Although skip pointer algorithm avoids visiting the already visited tree node, skip pointer may visit a large number of tree nodes if data objects are not stored preserving spatial locality. For example, suppose the leaf node D and G in Figure 2 have overlapping data objects. After processing node D, the skip pointer tree traversal algorithm will visit all the rest of the tree nodes, i.e, E, F, C, G, H, I, and J. Note that the skip pointer transforms the hierarchical tree structure into sequential memory block in depth first order. Hence, even if tree node D does not overlap, node D must be accessed in order to access its right sibling node E in skip pointer traversal. As we will show in the experiments section, skip pointer works quite well when selection ratio is high and the degree of tree nodes is small, and when data points are clustered and stored having spatial locality in tree structures.

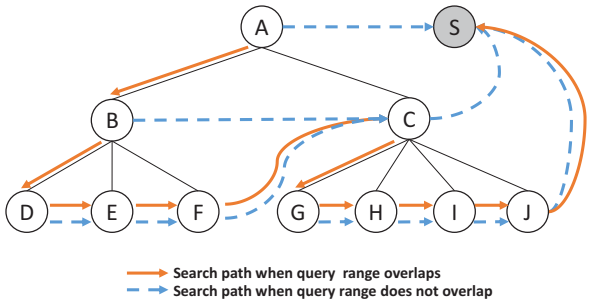


Fig. 2. *Query Processing with Skip Pointer*

3.2.6 Short Stack for R-tree

Horn et al. [12] extended Foley's kd-restart algorithm by employing a stack of bounded size. Pushing a new node onto stack will delete the node at the bottom of stack. When a tree traversal backtracks, it first searches the short stack. If the short stack is not empty, the parent node can be visited by accessing the topmost node on the stack. If the stack is empty, it restarts the search operation at the root of the tree again as in kd-restart.

Since the short stack algorithm can be used for n-ary tree structures and range query processing, we implemented the short stack algorithm for parallel R-trees and multi-way bounding volume hierarchies. For multi-dimensional range queries, the short stack helps reduce the number of global memory accesses compared to a parent link and skip pointer. However, as we will show, it incurs non-ignorable amount of overhead due to stack operations. Also, the stack miss ratio increases when the degree of tree node is large and tree height is tall.

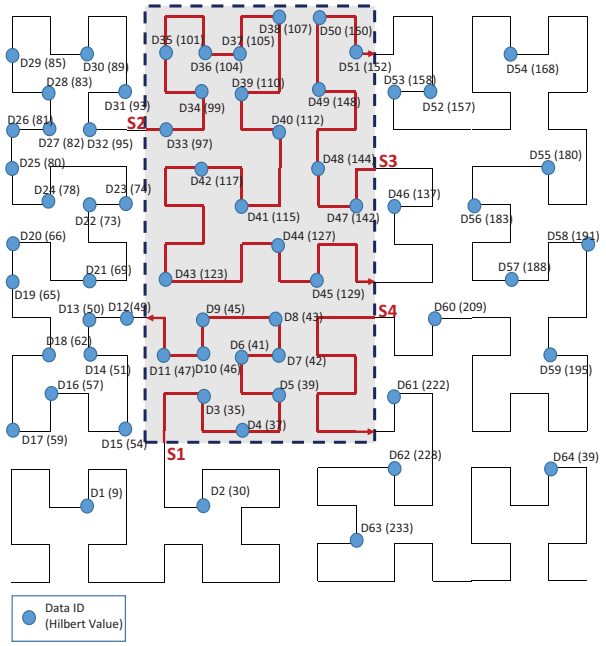


Fig. 3. *Hilbert Curve: Multi-dimensional range query overlaps some number of runs (S1∼S4) on Hilbert curve, i.e., the data objects that overlap a multi-dimensional query range are discontinuously stored.*

4 Massively Parallel Processing of N-ary Multi-dimensional Index

In scientific data analysis applications, the size of data sets is usually enormous but the number of submitted concurrent queries is relatively smaller than enterprise database systems and ray tracing in computer graphics. Hence in this work we focus on reducing the query response time and extending n-ary multi-dimensional indexing structures - R-tree, so that multiple GPU threads can cooperate in order to process a single query in parallel.

4.1 Massively Parallel Hilbert R-Tree

In order to avoid stack operations and recursion, we developed *Massively Parallel Hilbert R-tree* (*MPHR-tree*). MPHR-tree tags each leaf node with a sequential number - *leaf index* from left to right, and internal tree nodes of MPHR-tree store the *maximum* leaf index of its sub-trees as shown in Figure 3 and Figure 4. While traversing the tree structure, the query processing threads keep track of the largest leaf index that they have visited. The maximum leaf index stored in each tree node is used to avoid recursive back-tracking and re-visiting previously visited nodes. Instead of the sequential leaf index, Hilbert value of data objects can be used to allow dynamic insertion of data object into previously constructed MPHR-tree, but the Hilbert value usually requires larger amount of storage, and multiple data objects can be mapped to the same Hilbert value if the level of Hilbert curve is not fine-grained enough to distinguish all the data objects.

Scientific datasets are usually static, i.e, they do not change once they are acquired from sensor devices. Taking advantage of this, we sort the multi-dimensional data objects using a space filling curve - *Hilbert curve* that preserves good spatial locality [21]. As Hilbert curve clusters spatially nearby objects, we can create tight bounding boxes for the sorted datasets. With the bounding boxes, R-trees can be constructed in a bottom-up fashion as in *Packed R-trees* [16]. The bottom-up construction makes the node utilization of low level trees nodes almost 100%, however it may result in large overlapping regions for the bounding boxes in root node.

Parallel sorting on the GPU has been studied in many recent literature including [28]. In order to sort the Hilbert values of the multi-dimensional data, we employed *Thrust* [23], which is an open source C++ STL-like GPU library that implements many core parallel algorithms including radix sort. After sorting the entire data objects using the radix sort on the GPU, B number of consecutive data objects are stored in the same leaf node where B is the maximum number of data that the leaf node can hold. After assigning all the data objects to leaf nodes, the bottom-up constructed MPHR-tree builds MBRs of leaf nodes via parallel reduction and stores the MBRs in their parent nodes. After creating parent nodes, the bottom-up construction goes up one level, and repeats until only one root node is left. This construction can be easily parallelized on the GPU.

4.2 Multi-way Space Partitioning Bounding Volume Hierarchy

Alternatively, we can build a tree structure in a top-down fashion by recursively partitioning the datasets. This approach will reduce the overlap amount in high level tree nodes, but it may decrease the node utilization of leaf nodes. In order to eliminate any overlap between bounding boxes, we can employ binary space partitioning (*BSP*) or multi-way space partitioning (*MSP*) method. In MSP style partitioning, each split results in disjoint sub-spaces. However, since MSP takes less advantage of spatial locality compared to space filling curve, spatially nearby objects can be scattered across distant leaf nodes. In our experiments, we compare the stackless multi-dimensional range query processing algorithms using both the bottom-up constructed MPHR-tree and the disjoint multi-way MSP BVH.

4.3 MPRS: Massively Parallel Restart Scanning

Massively Parallel Restart Scanning (*MPRS*) algorithm is a multi-dimensional range query processing algorithm we propose which traverses the hierarchical tree structures from root node to leaf nodes multiple times as in kd-restart algorithm [7]. For a given range query, no matter how many MBRs of child nodes overlap a given query range, our MPRS algorithm always selects the leftmost overlapping child node unless all its leaf nodes have been already visited. Once it determines which child node to access in the next level, it does not store the overlapping child node information in the current tree node as an activation record but immediately discards it because our MPRS algorithm does not backtrack to already visited tree nodes.

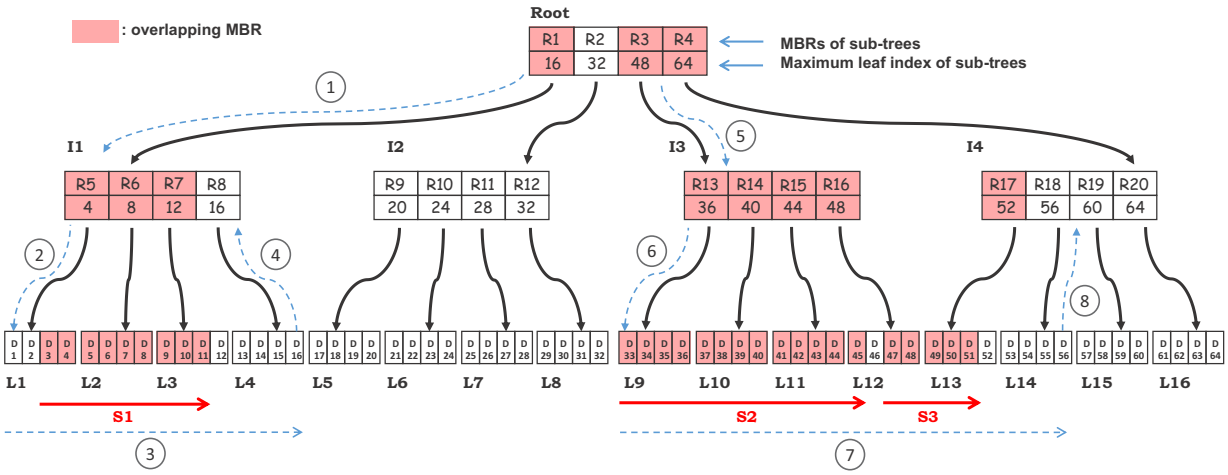


Fig. 4. *Massively Parallel Restart Scanning with MPHR-tree Structure*

The MPRS range query algorithm resembles the B+-tree search algorithm in that both search algorithms scan leaf nodes. In one-dimensional B+-tree, range query traverses hierarchical trees to find out the leftmost (smallest) data value within the query range, and performs leaf level scanning until it finds out a data value greater than query range and terminates the search. However, in multi-dimensional space, data objects that overlap a given query range may not be located in a single span of leaf nodes. Even after sorting the multi-dimensional data objects using the Hilbert space filling curve, a query range may overlap leaf nodes in multiple segments on the Hilbert curve. For example, a range query illustrated as a shaded box in Figure 3 overlaps four Hilbert curve segments - $S1$, $S2$, $S3$, and $S4$.

The MPRS search algorithm finds the smallest (in terms of the Hilbert curve index) multi-dimensional data object that overlaps a query range ($D3$ in Figure 3). After it finds out an overlapping data object that has the smallest Hilbert value, it starts scanning its next sibling data objects to find out if they also overlap. Due to the spatial locality property of Hilbert space filling curve in multi-dimensional space, it is highly likely that the right sibling data objects also overlap and they could be on the same continuous segment on the Hilbert curve ($D4 \sim D11$). While scanning data objects along the continuous segment on the Hilbert curve, it needs to jump to the starting position ($D33$) of the next segment ($S2$) on the Hilbert curve if it visits a non-overlapping data object ($D12$).

Our MPRS search algorithm scans and compares the overlap of a given query with data objects on the continuous Hilbert segment in a massively parallel way using a large number of threads on the GPU. If any single thread finds an overlapping data object, the scanning keeps fetching the next group of data objects and compares the overlap. However while scanning data objects on the Hilbert curve, all threads may find out none of the data objects are in the query range. If so, it stops scanning leaf nodes and restarts traversing MPHR-tree to find out the starting point of the next Hilbert curve segment that overlaps the query range.

When restarting the tree traversal, MPRS search algorithm uses the leaf index stored in tree nodes in order to avoid visiting already visited leaf nodes. In the restarted tree traversal, any tree node whose maximum leaf index is smaller than the maximum leaf index of previously visited leaf nodes is ignored. This is simply because if we have visited a leaf node v there's no reason to visit internal tree nodes which are parent nodes of v's left siblings. In each restart traversal, only if a tree node overlaps a query and it is the leftmost child node that has at least one unvisited leaf node, the tree node is accessed in the next level.

If all the overlapping data objects are stored in a single span of consecutive leaf nodes, the root node is accessed only once, which is the best case. Due to the clustering property of Hilbert curve, it is unlikely that the overlapping leaf nodes are widely spread throughout a large number of leaf nodes interleaved by non-overlapping leaf nodes. However, there might still be a chance that some nearby data objects can be spread across many non-contiguous sections of a Hilbert curve. In such a case, multiple restart tree traversal is necessary to skip a large number of non-overlapping sections of the Hilbert curve.

In order to reduce the number of restart tree traversal, we employ minimal backtracking, i.e, instead of starting a new tree traversal from root node immediately after visiting a non-overlapping leaf node, our MPRS algorithm fetches a parent of the last visited leaf node from global memory. In the parent node, it checks if it has any other leaf node that overlaps the query range and has a leaf index higher than the maximum leaf index of previously visited leaf nodes. If the parent node does not have such an overlapping leaf node, MPRS algorithm starts another tree traversal from root node. If the parent node has an overlapping but unvisited leaf node, leaf node scanning continues from the leaf node. In our experiments, the parent node check reduces the number of restart tree traversal by $20 \sim 27\%$.

The details of the MPRS range query algorithm are described in Algorithm 1. `MPRS_RangeQueryKernel()` is the kernel function that processes a single multi-dimensional range query in parallel. A block of threads fetches a tree node from global memory and each thread compares a query range

Algorithm 1 *MPRS Range Query Algorithm*

```
void MPRS_RangeQueryKernel(Node* root, MBR q)
1: shared MBR query ← q;
2: shared int Ovlp[NumberOfChildNodes];
3: int visitedLeafIdx ← 0
4: parfor tid ← 1, numThreads do
5:     node ← root
6:     // Iterate until tree traversal is done
7:     while visitedLeafIdx < root.maxLeafIdx do
8:         // Visit a new tree node
9:         while node is an internal node do
10:            Ovlp[tid] ← INT_MAX
11:            // each thread compares a query with the
12:            // bounding box of its child node
13:            if tid < node.numChildren and
14:             visitedLeafIdx < node.leafIdx[tid] and
15:             Overlap(query, node) then
16:                // If an overlapping child node is unvisited
17:                Ovlp[tid] ← tid
18:            end if
19:            syncthreads()
20:            // parallel reduction to find out the
21:            // leftmost overlapping child
22:            leftmost ← parallelReduction(Ovlp)
23:            if leftmost == INT_MAX then
24:                // If there's no overlapping child node,
25:                // update the visitedLeafIdx
26:                visitedLeafIdx ← node.maxLeafIdx
27:                // restart the search from root
28:                node ← root;
29:            else
30:                // fetch the leftmost child node
31:                node ← node.child[leftmost]
32:            end if
33:            syncthreads()
34:        end while
35:        while node is a leaf node do
36:            if tid < node.numChild and
37:             Overlap(query, node) then
38:                hitFlag ← true
39:                SaveOverlappingData(node.data[tid])
40:            end if
41:            visitedLeafIdx ← node.maxLeafIdx
42:            if hitFlag == true then
43:                // fetch next sibling node
44:                node ← node.rightSibling
45:            else
46:                // parent check
47:                node ← node.parent
48:            end if
49:        end while
50:    end while
51: end parfor
```

with a bounding box of a child node in parallel. After storing the overlap flags in shared memory array ($Ovlp$), the leftmost and unvisited overlapping child node is identified by parallel reduction and the child node is fetched from global memory. If none of the child nodes overlap the query range, the largest leaf index of its sub-tree replaces the current maximum leaf index of visited tree nodes - $visitedLeafIdx$ and it starts tree traversal from root node again. The $visitedLeafIdx$ is compared against each child node's maximum leaf index, and if $visitedLeafIdx$ is larger than a child node's maximum leaf index, the child node is ignored.

4.3.1 Example: Massively Parallel Restart Scanning

Figure 4 shows an example of MPHR-tree and the MPRS search path for the data objects and the query illustrated in Figure 3. In the beginning, $visitedLeafIdx$ is set to 0 and each thread compares the MBR of each child node with a query range. Suppose the query range overlaps MBRs - $R1$, $R3$, and $R4$ in the root node. The MPRS algorithm ignores $R3$ and $R4$, and visits the leftmost child node $I1$ (①). Again, the query is compared with the MBRs in the node $I1$, and the leftmost overlapping leaf node $L1$ is selected (②). Once we reach a leaf node, the multi-dimensional coordinates of the data objects in the leaf node $L1$ are compared with the given query range. If some of the data objects overlap in a leaf node, its right sibling leaf node $L2$ will be visited and checked for overlapping data (③). Note that it is trivial to compute the memory address of a right sibling node since when we construct MPHR-trees the leaf nodes are stored in a big contiguous memory block, thus we can simply add the constant size of a tree node to the starting address of a current node. With such a contiguous memory block, scanning leaf nodes can take advantage of CUDA memory access optimization techniques such as *memory coalescing* which gives a big performance gain. In the example, MPRS algorithm keeps visiting right sibling leaf nodes $L3$ and $L4$. However because $L4$ has no overlapping data objects, its right sibling leaf node $L5$ is not accessed but we check its parent node $I1$ (④). Note that $I1$ was a previously visited node in this example, but note that parent check may visit an unvisited internal tree node if an overlapping section of Hilbert curve is long. Since $I1$ does not have any other overlapping child node which was never visited, MPRS algorithm restarts the search from root node.

When leaf node scanning stops, the $visitedLeafIdx$ is set to 16 that is the largest leaf index stored in node $I1$. In the next restart traversal, although $R1$ overlaps the query range, $R1$'s leaf index 16 is not greater than the current $visitedLeafIdx$, thus thread 1 ignores $R1$. Thread 2 also ignores $R2$ since its MBR does not overlap the query range. Thread 3 detects the overlap between $R3$ and the query, and since its leaf index 48 is greater than the current $visitedLeafIdx$, $I3$ will be selected as the next child node to visit (⑤). Thread 4 will also find its MBR $R4$ overlaps, but $I4$ will not be accessed since it is not the leftmost overlapping child node in current tree traversal. In $I3$, $L9$ will be selected as the child node to visit (⑥). In $L9$, data objects that overlap the query - $D33$, $D34$, $D35$, and $D36$ are found. Thus, its right sibling nodes $L10$, $L11$, $L12$, $L13$, and $L14$ are scanned and the $visitedLeafIdx$ will be updated to 56 (⑦). Since $L14$ does not have any

overlapping data object, parent check optimization fetches its parent node $I4$ from global memory. In node $I4$, $R17$ is the only MBR that overlaps but its leaf index 52 is smaller than the current $visitedLeafIdx$ 56, hence it returns after setting $visitedLeafIdx$ to 64 (⑧). In the next round of restart, a block of threads do not find any overlapping child node in the root node that has a leaf index value greater than 64. Finally, the search kernel function returns and the search finishes.

4.3.2 Analysis of Massively Parallel Restart Scanning

The complexity of the MPRS range query algorithm is as follows:

Theorem 1 The search complexity of the MPHR search algorithm is $O(C \times log_B N + k)$, where C is the number of Hilbert curve segments that overlap a given multi-dimensional query range, N is the number of data objects stored in the tree, B is the number of degree of tree nodes, and k is the number of leaf nodes that contain data objects within query range.

Proof: The number of restarting tree traversal is C - the number of spans of contiguous overlapping level $height - 1$ nodes (parents of leaf nodes) in MPHR-trees. Each tree traversal from a root node to a leaf node visits a single tree node in each tree level and the height of the MPHR-tree is $log_B N$, thus $C \times log_B N$ tree nodes are accessed to find unvisited leftmost overlapping leaf nodes in total.

For each identified unvisited leftmost leaf node, leaf scanning visits some number of right sibling leaf nodes. Since the sibling leaf nodes are accessed only if they have overlapping data objects, the total number of visited leaf nodes is k. Note that the k is determined by selection ratio of range query. □

C can be as large as $\lceil N/B^2 \rceil/2$ in the worst case if there exist no two or more consecutive overlapping nodes in level $height - 1$, i.e., an overlapping level $height - 1$ node and a non-overlapping level $height - 1$ node take turns. In such a case, whenever an overlapping level $height - 1$ node is found by tree traversal from root node, its adjacent non-overlapping right sibling node will trigger another tree traversal from root node. Hence, $\lceil N/B^2 \rceil \times 1/2$ times tree traversal from root node will be needed in that case. Then, the total number of visited tree nodes in the worst case is $O(\lceil N/B^2 \rceil \times 1/2 \times log_B N + k)$, and all the level $height - 1$ nodes will be visited.

In [13], the maximum number of continuous runs on the Hilbert curve for a given multi-dimensional range query is analyzed, but the number of consecutive level $height - 1$ nodes in MPHR-trees can not be determined by the number of continuous runs if the data objects are not uniformly distributed on the Hilbert curve. Thus, C can be as large as $\lceil N/B^2 \rceil/2$ in the worst case, but note that in such a worst case, the recursive R-tree search algorithm will also visit all the higher level tree nodes and half of the level $height - 1$ nodes ($\lceil N/B^2 \rceil \times 1/2$). In our experiments with real scientific datasets, the number of restart tree traversal (C) rarely exceeds 30 when the data set consists of 32 million three dimensional objects.

5 Experiments

5.1 Experimental environment

In this section, we evaluate and analyze the performance of stackless range query processing algorithms on the GPU. We conduct the experiments on a CentOS Linux machine that has quad AMD Opteron 8 Core 6128 2.0GHz processors (NUMA system) and 64 GB DDR3 memory with NVIDIA Tesla M2090 GPU which has 512 CUDA cores that can host maximum 1536 resident threads. We use CUDA 5.5 for all the experiments.

The bottom-up construction method of the MPHR-tree described in Section 4 makes the node utilization of most low level trees nodes 100%, however it may result in large overlapping regions for the bounding boxes in upper level nodes. The large overlapping region is known for the main cause of poor multi-dimensional range query performance. In order to eliminate any overlap between bounding boxes, we implemented disjoint space partitioning bounding volume hierarchy - multi-way MSP bounding volume hierarchy, which is similar to K-D-B-tree [27]. It should be noted that spatially nearby objects in our n-ary MSP bounding volume hierarchy can be scattered across highly distant leaf nodes since it does not take advantage of a space filling curve. Using the MPHR-tree and the MSP multi-dimensional indexing trees, we compare the performance of stackless multi-dimensional range query processing algorithms - skip pointer, parent link, short stack, and MPRS described in Section 3.2 and 4.

To evaluate the MPHR-trees and the MSP BVH with the stackless range query processing algorithms, we index three dimensional *Integrated Surface Database* (*ISD*) point datasets available at 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 by over 20,000 stations such as wind speed and direction, temperature, pressure, precipitation, etc. For the experiments, we index 40 million values, each consists of latitude, longitude, time, and a pointer to the sensor value, collected from the year 2010 to 2012. With the three dimensional ISD datasets, we synthetically generate five sets of 160,000 queries with various selection ratio, which determines the range of queries and how many data objects overlap the range of a given query. We also evaluate the indexing methods with other real datasets including remotely sensed AVHRR (Advanced Very High Resolution Radiometer) GAC (Global Area Coverage) level 1B datasets, but do not present their results since the results are similar with the presented analysis. In addition to the real datasets, we synthetically generate 64 million point and rectangular datasets in uniform, normal, and Zipf's distribution in order to evaluate the indexing schemes in high dimensional spaces, but we only present the results of uniform distribution since the comparative performance results with the other distributions are similar.

As for the performance metrics, we measure the average query response time that is the average time for the GPU kernel function to return the search results back to the CPU host for a single search query. We also measure *the warp execution*

efficiency using NVIDIA profiler to show SIMD efficiency, and the number of visited tree nodes since the number of visited tree nodes is the most important performance factor that determines the query response time.

5.2 Experimental results

5.2.1 Parallel Construction of MPHR-trees

TABLE 1
Construction Time of Massively Parallel Hilbert R-trees on Tesla M2090

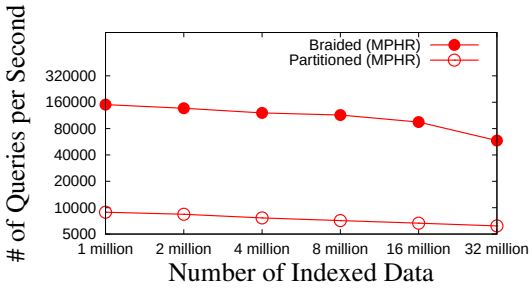
Time (sec)	Number of Inserted Data (millions)					
	2	4	8	16	32	40
Sorting	0.138	0.295	0.582	1.110	2.299	2.866
Memory Transfer	0.036	0.070	0.139	0.277	0.552	0.689
MPHR-tree Construction	0.003	0.005	0.011	0.021	0.042	0.053
Total Time	0.177	0.370	0.732	1.408	2.893	3.608

It must be noted that the performance of index construction is not as important as the performance of query processing in scientific data analysis applications since scientific datasets do not change once they are stored and the constructed index does not have to be rebuilt. Even though the index creation is not the dominant operation in processing scientific datasets and the parallel bottom-up index construction is not the main contribution of this work, we measure the parallel index construction performance because the performance of constructing packed R-tree in parallel *on the GPU* has not been reported in literature to the best of our knowledge and the cost of building the tree index on the CPU is very high compared to a search cost.

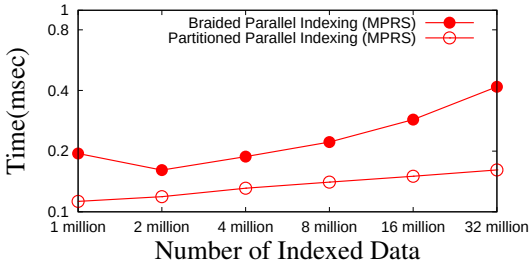
Table 1 shows the elapsed time of constructing MPHR-trees in a bottom-up fashion on a single NVIDIA M2090 GPU. In the experiments, the MPHR-tree construction spends more than 77% of its total construction time (2.866 seconds for 40 millions of data) on sorting Hilbert values, while the bottom-up parallel construction of the MPHR-trees takes only about 1.4% of its total construction time (0.053 seconds) [1] Overall, it takes less than 4 seconds to construct a multi-dimensional index for 40 millions of data on an NVIDIA M2090 GPU.

5.2.2 Braided Parallelism

In order to maximize the utilization of GPU cores, which plays a key role in improving the query response time and query processing throughput, we compare two different parallel index search approaches on the GPU. In GPU computing, *braided parallelism* implies that multiple jobs run concurrently on different SMs, and multiple GPU cores in each SM concurrently process a single job in a data parallel fashion [6]. The braided parallelism is commonly used in high performance GPU applications to maximize throughput as well as to improve the execution time. In braided parallel indexing, a single MPHR-tree or a single MSP BVH in global memory is shared by multiple SMs, and each SM accesses different parts of the tree to process its own query. The other approach we compare is the *data parallelism*. In order to maximize the data parallelism, we developed partitioned indexing. In partitioned indexing, a single MPHR-trees is partitioned into multiple sub-trees, and a single query is processed by all the SMs that access their own partitioned trees. The partitioned indexing can help even further decreasing the query execution time of a single query since the amount of work is reduced and spread across more processing units.



(a) Query Processing Throughput



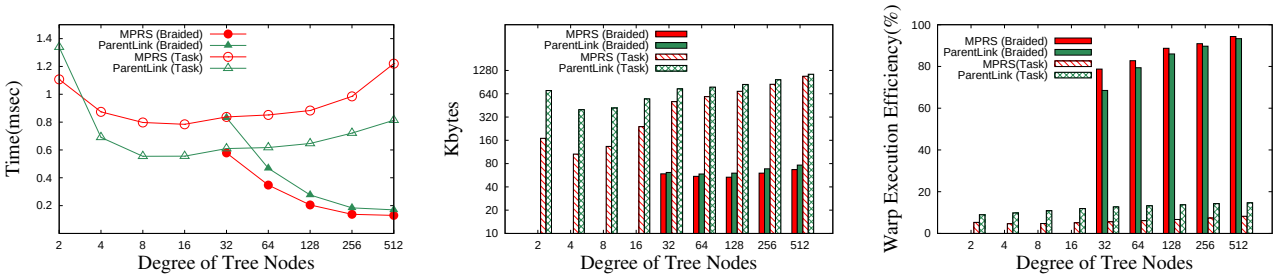
(b) Average Query Response Time

Fig. 5. *Query Processing Performance With Braided Parallel Index and Partitioned Index*

In the experiments shown in Figure 5, we used the braided parallel MPHR-tree indexing and data parallel partitioned MPHR-tree indexing to measure the query processing throughput and average query response time with varying the number of indexed data objects. In terms of the average query execution time, the partitioned indexing exhibits about 45%∼160% faster performance than the braided parallel indexing, however the braided parallel indexing processes 9.4∼16.8 times larger number of range queries than the partitioned indexing. Since the braided parallel indexing shows much higher query processing throughput while its query response time is slightly slower than the braided parallel indexing, we use braided parallel indexing for the rest of the experiments.

The maximum number of resident blocks per SM in an NVIDIA M2090 GPU is eight, and the number of SMs in a M2090 GPU is 16, thus the best query processing throughput is achieved with 128 (16x8) thread blocks in our experiments. For the rest of the experiments that use braided parallel indexing, the number of CUDA thread blocks for the index search kernel function is fixed to 128, i.e., 128 concurrent

1. On AMD Opteron 6128 CPU, it takes 1.72 seconds to construct the MPHR-trees with 40 million data objects in a bottom-up fashion without including sorting time, which is 32 times slower than bottom-up construction on the GPU. Also, note that the Hilbert value sorting on AMD Opteron 6127 CPU takes about 10 seconds to sort 40 millions of data.



(a) Average Query Execution Time per Query (b) Global Memory Access Per Query (c) SIMD Efficiency

Fig. 6. *Search Performance With Varying the Degree of Tree Nodes*

queries can be processed in parallel on a single GPU using braided parallel indexing.

5.2.3 Varying Degree of Tree Nodes

In the experiments shown in Figure 6, we vary the degree of tree nodes (the number of child nodes) and measure the average query response time and the total amount of global memory accesses. When the degree is B, the braided parallel indexing lets B GPU threads access the same tree node, and the B bounding boxes in the node are concurrently compared against a given query range. For braided parallel indexing, we didn't measure the average query execution time for smaller degrees than 32 since smaller degrees need fewer number of CUDA cores than the warp size and it will hurt the CUDA core utilization (SIMD efficiency). When the degree is a multiple of warp size, the query execution time with parent link and MPRS algorithm improves but other stackless algorithms including skip pointer algorithm perform much worse. Hence we do not plot the performance of them due to the scale of the graph. As described in section 3.2.5, skip pointer needs to visit more leaf nodes as the degree of tree node increases even if they do not overlap a query range.

In addition to the braided parallel indexing, we implemented task parallel indexing schemes - *MPRS(Task Parallel) and ParentLink(Task Parallel)*, where each CUDA thread processes a different query as in ray tracing. In task parallel indexing, we let each SM process 32 queries in parallel no matter what the degree of tree node is. If the degree is k, each thread sequentially compares k MBRs with its own query. As we increase the degree of tree node, the tree height decreases and the number of accessed tree nodes per query also decreases, but since the size of a tree node increase with a larger degree, the total amount of global memory accesses per query increases as shown in Figure 6(b). Moreover a large degree of tree node does not help improving the average query execution time since the number of bounding boxes to compare with a given query per each tree node increases. If the degree of tree is too small, the size of tree node becomes smaller than L1 cache line size, hence binary tree may over-fetch unnecessary parts of global memory. As a result, binary tree accesses 77% more data than 4-ary trees in the experiments. However as the degree increases, the size of tree node grows and it results in accessing a larger amount of data from global memory.

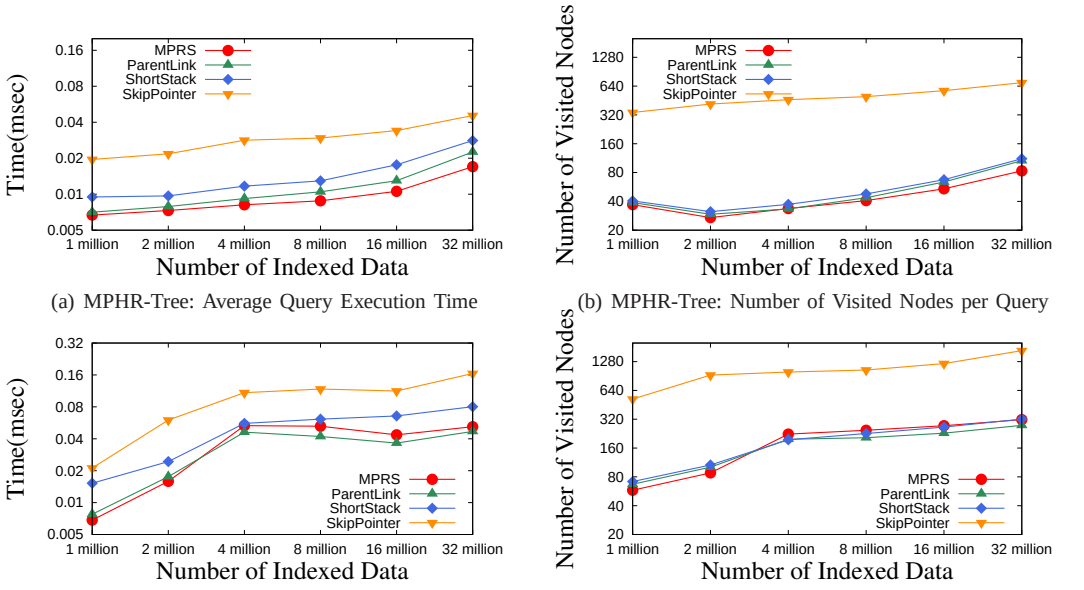
Figure 6(b) shows task parallel parent link algorithm accesses about 7∼20 times more data from global memory than braided parallel MPRS algorithm. This is because task parallel indexing schemes access scattered memory blocks as we described earlier and it ends up reading the same tree nodes multiple times due to warp divergence and L1 cache replacement.

Figure 6(c) shows the warp execution efficiency of braided parallel MPRS range query algorithm is much higher than that of task parallel algorithms. Warp execution efficiency is the ratio of the average active threads per warp to the maximum number of threads per warp, which indicates SIMD efficiency [1]. If each thread in a block processes a different range query, the number of tree node accesses per thread diverges in task parallelism, i.e., a query whose range is wider will visit more tree nodes than a query whose range is smaller. However, in braided parallel indexing, all threads in a block accesses the same tree nodes, hence its warp execution efficiency is determined by the tree node utilization. If all tree nodes are completely packed, the warp execution efficiency of braided parallel indexing will be almost 100%. As our MPRS range query algorithm visits mostly leaf nodes and it visits fewer number of internal tree nodes than parent link, warp execution efficiency of MPRS algorithm is consistently higher than 80% while parent link algorithm yields about 65% warp execution efficiency when the degree is 32.

5.2.4 MPHR-Trees vs. disjoint MSP BVH

In Figure 7, we compare the stackless multi-dimensional range query processing algorithms with two braided parallel indexing schemes - MPHR-tree shown in Figure 7(a) and disjoint MSP BVH shown in Figure 7(c). Overall, MPHR-tree consistently shows faster average query execution time than disjoint MSP BVH. When the number of indexed datasets is 32 millions, MPRS algorithm with MPHR-tree processes range queries 2.8 times faster than parent link algorithm with disjoint MSP BVH (0.016 msec vs. 0.046 msec).

When MPHR-tree indexes 32 million 3D data points, parent link algorithm exhibits the second fastest average query execution time per query (0.022 msec) and the skip pointer shows the slowest average query execution time (0.045 msec) which is 2.8 times higher query execution time than that of MPRS. In order to analyze the performance differences between the stackless range query processing algorithms, we measured the number of visited tree nodes per query. With 32 million of data points, the MPRS algorithm accesses 85 tree nodes from global memory per each query on average, and it restarts tree



(a) MPHR-Tree: Average Query Execution Time

(b) MPHR-Tree: Number of Visited Nodes per Query

(c) Disjoint MSP BVH: Average Query Execution Time

(d) Disjoint MSP BVH: Number of Visited Nodes per Query

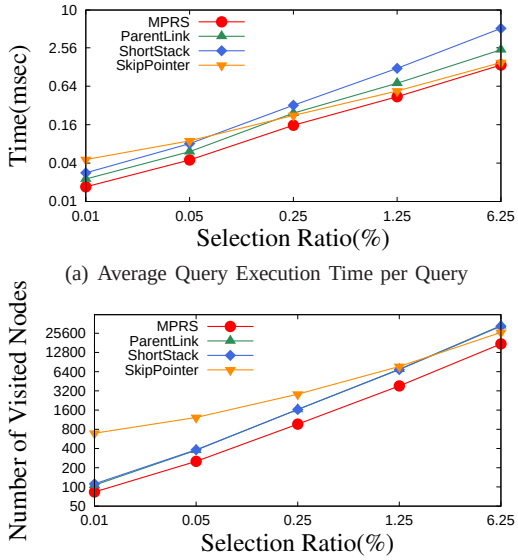
Fig. 7. *Average Query Response Time With MPHR-Tree and disjoint MSP BVH*

traversal 9 times. Short stack algorithm accesses 63 tree nodes from global memory, pushes 4 tree nodes onto short stack and reads 54 tree nodes from the short stack in shared memory. Although the number of push operations is small, the overhead of push operation is much higher than reading small necessary part of a tree node from global memory since push operation reads an entire tree node from global memory and stores it in shared memory. Although reading shared memory is faster, it still causes some I/O overhead, and as a result, the query execution time with short stack algorithm is higher than that of MPRS algorithm. Parent link algorithm accesses about 108 tree nodes from global memory, and skip pointer algorithm visits about 690 tree nodes, which is about 1.3 times and 8.1 times higher number than the number of global memory accesses of MPRS respectively.

5.2.5 Selection Ratio

In the experiments shown in Figure 8, we varied the selection ratio of queries with the MPHR-tree. As the selection ratio grows, a larger number of leaf nodes must be visited and the number of visited nodes increases almost linearly as the selection ratio increases. As a result, the query response time increases and the query processing throughput decreases.

When the selection ratio is 0.01%, i.e., a query retrieves 4,000 data objects from the indexed 40 million data objects, the MPRS algorithm accesses only 84 tree nodes taking 0.017 msec while the second best parent link algorithm visits 108 tree nodes taking 0.023 msec. As the selection ratio increases, the performance gap between the MPRS and the parent link algorithm grows because a larger number of overlapping data objects are stored in contiguous leaf nodes. When the selection ratio is 6.25%, the parent link algorithm (2.38 msec) takes 1.73 times longer than the MPRS (1.37 msec) algorithm. Interestingly, skip pointer algorithm shows comparable performance (1.50 msec) to MPRS algorithm when selection ratio is high. However, when selection



(a) Average Query Execution Time per Query

(b) Number of Visited Tree Nodes Per Query

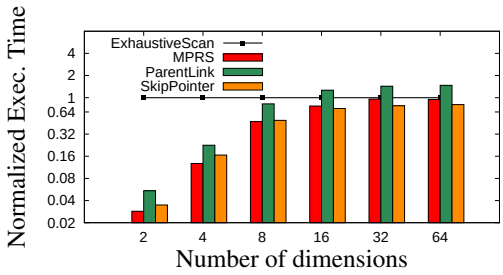
Fig. 8. *R-Tree: Search Performance with Varying Selection Ratio*

ratio is small, the skip pointer visits 8.3 times more tree nodes than MPRS and exhibits the worst performance. On the contrary, as the selection ratio increases, short stack suffers from low data reuse ratio and exhibits 2.75 times slower query execution time than MPRS.

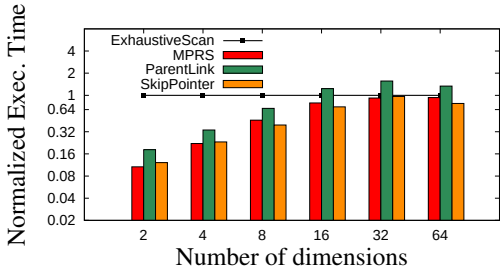
5.2.6 Performance in High Dimensions

For the last set of experiments shown in Figure 9, we measure the search performance with varying the number of dimensions of synthetic datasets. Although the dimension of the real-world scientific datasets is hardly bigger than four for practical reasons, the number of dimensions is known as one of the important performance factors to evaluate multi-dimensional

indexing structures. [2]. In order to generate synthetic 64 million high dimensional data objects and 1000 queries of which selection ratio is 1%, we used a random number generator.



(a) MPHR-Tree: Average Query Execution Time per Query



(b) MSP BVH: Average Query Execution Time per Query

Fig. 9. *Search Performance with Varying Number of Dimensions*

In high dimensions, it is well known that hierarchical multi-dimensional indexing trees do not perform well because the volume of the space grows exponentially. In R-tree, the amount of overlap between the minimum bounding boxes of upper level tree nodes exponentially increases. Even for the disjoint MSP BVH, a large number of sub-partitions are likely to overlap a query range in high dimensions because 40 million data points do not need more than 26 splits. Hence the ratio of pruning sub-trees is usually very low in high dimensions. This is the well-known *curse of dimensionality* problem [2]. In high dimensions (> 64), a brute-force exhaustive scanning of all the data objects often performs faster than hierarchical tree structured indexing. The brute-force exhaustive scanning can effectively utilize a large number of processing units on the GPU, hence we compare the performance of the stackless tree traversal algorithms against brute-force *ExhaustiveScan* which is the performance baseline.

Since the ExhaustiveScan scans the entire indexed data objects, the number of accessed data nodes is independent of the dimensions. Note that the ExhaustiveScan has only data nodes without hierarchical structure. For the experiments shown in Figure 9, we indexed 64 million data points in two dimensions, but as we increase the dimensionality, we reduce the number of indexed data objects since the global memory size of Tesla M2090 GPU is limited to 6 GBytes. I.e., in 4 dimensions, we can index only 32 million data points, and in 64 dimensions, we index 2 million data points. Therefore, the number of visited nodes decreases linearly as the number of dimensions increases. When the dimension is smaller than 8, most of the stackless tree traversal algorithms with MPHR-tree outperforms the brute-force exhaustive scanning. But when the dimension is higher than 8, the brute-force ExhaustiveScan shows comparable performance to the MPRS algorithm and it's even faster than short stack and parent link algorithm.

This experimental result also confirms that MPRS algorithm and MPHR-tree do not perform well in high dimensions as with other tree structured indexing schemes. As shown in Figure 9(b), although MSP BVH does not have overlapping region between tree nodes, it does not work well in high dimensions for the same reason why K-D-B-tree does not perform well in high dimensions.

With the 64-dimensional datasets and 1% selection ratio range queries, the MPRS search algorithm visits 77% of tree nodes in MPHR-trees. Interestingly, the performance gap between skip pointer and MPRS algorithm decreases as the dimension increases, and the skip pointer outperforms the MPRS algorithm when the dimension is higher than 16. And skip pointer outperforms parent link algorithm since the selection ratio is 1%. Note that Figure 9(a) shows the skip pointer works faster than parent link when the selection ratio is 1%, but still MPRS works faster even when the selection ratio is as high as 6.25%. If selection ratio is high and data points are in high dimensions, skip pointer keeps visiting sibling leaf nodes similar to the brute-force exhaustive scanning, which explains its relatively good performance in high dimensions.

In summary, the MPRS algorithm consistently outperforms all the other indexing schemes in terms of throughput and query execution time in dimensions lower than 16. However the performance gap decreases as the dimension increases.

6 Conclusion

In this work, we proposed a novel parallel multi-dimensional indexing structure, *MPHR-trees* and *MPRS* tree traversal algorithm for multi-dimensional range query processing on the GPU. It has been known that multi-dimensional indexing structures are not well suited to parallel systems due to recursion and irregular tree access patterns. MPRS tree traversal algorithm (i) uses a large number of GPU threads to process a single query in a SIMD fashion in order to improve the query execution time, (ii) avoids warp-divergence by fetching only a single tree node in each step for a block of threads in a streaming multiprocessor, (iii) avoids recursion or stack operations by restarting tree traversal and avoiding visiting previously visited tree nodes by tracking the largest leaf index of visited tree nodes, (iv) and accesses mostly contiguous memory block by leaf node scanning.

We also extended several stackless ray tracing algorithms - short stack, parent link, and skip pointer for multi-dimensional range query with n-ary indexing trees, and conducted comparative performance study and showed our MPRS range query processing algorithm outperforms the other stackless tree traversal algorithms mainly because our MPRS algorithm accesses mostly sequential memory blocks and does not backtrack to previously visited tree nodes.

2. For high dimensional datasets, k-NN nearest neighbor query seems to be a more interesting and important problem than orthogonal range queries. The MPRS range query algorithm can be easily adapted to handle k-NN queries based on the min-max distance, but it is out of scope of this work.

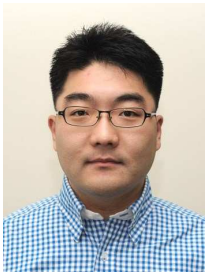
7 Acknowledgment

We would like to thank anonymous reviewers for their suggestions on early drafts of this paper. The corresponding author of this paper is Beomseok Nam. This research was supported by MKE/KEIT (No.10041608).

References

[1] NVIDIA CUDA Compute Unified Device Architecture. *http://www.nvidia.com/*.
[2] R. E. Bellman. *Adaptive Control Processes: A GuidedTour*. Princeton University Press, NJ, 1961.
[3] J. Chou, K. Wu, O. Rubel, M. Howison, J. Q. Prabhat, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani. Parallel index and query for large scale data analysis. In *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.
[4] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte. Medical image processing on the GPU – past, present and future. *Medical Image Analysis*, 17(8):1073 – 1094, 2013.
[5] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
[6] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided B+ tree searches on a GPU with cuda. In *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*, 2011.
[7] T. Foley and J. Sugerman. KD-tree acceleration structures for a gpu raytracer. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005.
[8] M. Folk. A White Paper: HDF as an Archive Format: Issues and Recommendations, January 1998. *http://hdf.ncsa.uiuc.edu/archive/hdfasarchivefmt.htm*.
[9] 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.
[10] M. Hapala, T. Davidoiv, I. Wald, V. Havran, and P. Slusallek. Efficient stack-less bvh traversal for ray tracing. In *the 27th Spring Conference on Computer Graphics (SCCG '11)*, 2011.
[11] V. Havran, J. Bittner, and J. Zara. Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics (SCCG)*, 1998.
[12] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *Symposium on Interactive 3D Graphics and Games (I3D)*, 2007.
[13] H. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1990.
[14] W.-K. Jeong and R. T. Whitaker. A fast iterative method for eikonal equations. *SIAM J. Sci. Comput.*, 30(5):2512–2534, 2008.
[15] T. Kaldewey, J. Hagen, A. D. Blas, and E. Sedlar. Parallel search on video cards. In *Proceedings of the First USENIX conference on Hot topics in parallelism (HotPar 09)*, 2009.
[16] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.
[17] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and 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.
[18] J. Kim, S. Hong, and 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.
[19] J. Kim and B. Nam. Parallel multi-dimensional range query processing with r-trees on GPU. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013.
[20] L. Luo, M. D. Wong, and L. Leong. Parallel implementation of r-trees on the GPU. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.
[21] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.
[22] B. Nam and A. Sussman. Improving access to multi-dimensional self-describing scientific datasets. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2003.
[23] NVIDIA. Thrust. *https://developer.nvidia.com/thrust*.
[24] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
[25] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Eurographics*, 2007.
[26] R. Rew, G. Davis, and S. Emmerson. NetCDF User's Guide for C, 1997. *http://www.unidata.ucar.edu/packages /netcdf/cguide.pdf*.
[27] J. T. Robinson. The K-D-B tree: A search structure for large multi-dimensional dynamic indexes. In *Proceedings of 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1981.
[28] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.
[29] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009.
[30] B. Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.
[31] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, Dec. 2007.
[32] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.

Jinwoong Kim is a graduate student in school of Electrical and Computer Engineeering at UNIST (Ulsan National Institute of Science and Technology) in Korea. He received his B.S. degree in computer science at Chungbuk National University, Korea in 2011. His research interests include parallel algorithms, GPGPU computing, and distributed and parallel systems.



Won-Ki Jeong is an assistant professor in school of Electrical and Computer Engineering at UNIST in Korea. From 2008 to 2011, he worked in School of Engineering and Applied Sciences at Harvard University as a research scientist. Jeong received his Ph.D. in Computer Science from the University of Utah in 2008. He is a recipient of NVIDIA graduate fellowship in 2007. His research interests include visualization, GPU computing, biomedical image processing, and computer graphics. He is a member of IEEE and ACM.

Beomseok Nam is an assistant professor in school of Electrical and Computer Engineering at UNIST in Korea. Nam received his Ph.D. in Computer Science in 2007 at the University of Maryland, College Park, and received a B.S. (1997) and an M.S. (1999) from Seoul National University, Korea. His research interests include data-intensive computing, database systems, and embedded system software. He is a member of IEEE, ACM, and USENIX.