



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

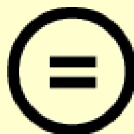
다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#) 

Doctoral Thesis

Exploiting Graphics Processing Units for Massively Parallel Multi-Dimensional Indexing

Jinwoong Kim

Department of Computer Science and Engineering

Graduate School of UNIST

2017

Exploiting Graphics Processing Units for Massively Parallel Multi-Dimensional Indexing

Jinwoong Kim

Department of Computer Engineering

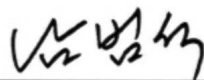
Graduate School of UNIST

Exploiting Graphics Processing Units for Massively Parallel Multi-Dimensional Indexing

A dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Jinwoong Kim

7. 5. 2017 of submission
Approved by



Advisor
Beomseok Nam

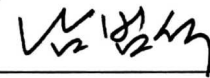
Exploiting Graphics Processing Units for Massively Parallel Multi-Dimensional Indexing

Jinwoong Kim

This certifies that the dissertation of Jinwoong Kim is approved.

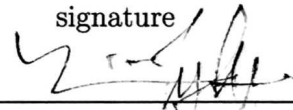
7/5/2017 of submission

signature



Advisor : Beomseok Nam

signature



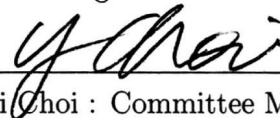
Sam H. Noh : Committee Member #1

signature



Won-ki Jeong : Committee Member #2

signature



Young-ri Choi : Committee Member #3

signature



Woongki Baek : Committee Member #4

ABSTRACT

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 for the past couple of decades.

Since the GPU has emerged as a new cost-effective performance accelerator, now it is common to leverage the massive parallelism of the GPU in various applications such as medical image processing [2], computational chemistry [3], and particle physics [4].

However, hierarchical multi-dimensional indexing structures are inherently not well suited for parallel processing because their irregular memory access patterns make it difficult to exploit massive parallelism [5]. Moreover, recursive tree traversal often fails due to the small run-time stack and cache memory in the GPU [6].

First, we propose *Massively Parallel Three-phase Scanning (MPTS)* R-tree traversal algorithm to avoid the irregular memory access patterns and recursive tree traversal so that the GPU can access tree nodes in a sequential manner. The experimental study shows that MPTS R-tree traversal algorithm consistently outperforms traditional recursive R-Tree search algorithm for multi-dimensional range query processing.

Next, we focus on reducing the query response time and extending n-ary multi-dimensional indexing structures - R-tree, so that a large number of GPU threads cooperate to process a single query in parallel. Because the number of submitted concurrent queries in scientific data analysis applications is relatively smaller than that of enterprise database systems and ray tracing in computer graphics. Hence, we propose a novel variant of R-trees *Massively Parallel Hilbert R-Tree (MPHR-Tree)*, which is designed for a novel parallel tree traversal algorithm *Massively Parallel Restart Scanning (MPRS)*. The MPRS algorithm traverses the MPHR-Tree in mostly contiguous memory access patterns without recursion, which offers more chances to optimize the parallel SIMD algorithm. Our extensive experimental results show that the MPRS algorithm outperforms the other stackless tree traversal algorithms, which are designed for efficient ray tracing in computer graphics community.

Furthermore, we develop query co-processing scheme that makes use of both the CPU and GPU. In this approach, we store the internal and leaf nodes of *upper tree* in CPU host memory and GPU device memory, respectively. We let the CPU traverse internal nodes because the conditional branches in hierarchical tree structures often cause a serious warp divergence problem in the GPU. For leaf nodes, the GPU scans a large number of leaf nodes in parallel based on the selection ratio of a given range query. It is well known that the GPU is superior to the CPU for parallel scanning. The experimental results show that our proposed multi-dimensional range query co-processing scheme improves the query response time by up to 12x and query throughput by up to 4x compared to the state-of-the-art GPU tree traversal algorithm.

DEDICATION

To my parents, Ilgyu Kim and Kyungsuk Park for their unconditional love and support.

Acknowledgments

I would like first to thank my advisor, Beomseok Nam who has encouraged me in all the time of research and guided me on the right track with his penetrating insight to all. It was an honor and very lucky to work with him. His mentorship and guidance have been essential in helping me become a researcher. I owe a lot to him for my professional and personal development and could not have had a better advisor.

I also would like to thank my thesis and oral examination committees, Professor Sam H. Noh, Professor Won-ki Jeong, Professor Young-ri Choi, and Professor Woongki Baek, for their help and valuable comments.

I thank my fellow labmates, Deukyeon Hwang and Wook-Hee Kim, for the stimulating discussions, and for all the fun we have had in the last seven years. Also, I am grateful to my friends in CISSR group for spending their spare time with me.

I am grateful to my close friends, Jaewuk Lee, Jinsoo Park, Kihyeon Kim, Jonghwan Yoon, and Jimin Chae, who have made me bearable in graduate school by chatting, drinking, playing, or whatever, deserve all the credit for making me keep working.

Finally, last but by no means least, I would like to thank my family, Ilgyu Kim, Kyungsuk Park, Jinwon Kim, and Yeojin Song. They always supported me and encouraged me when I was lost. They have made all of this happen.

Contents

Chapter 1. Introduction	1
1.1 Thesis and Contributions	3
1.2 Thesis Organization	5
Chapter 2. Related Work	6
Chapter 3. Multi-dimensional Range Query Processing with R-Tree on the GPU	9
3.1 Background on CUDA	9
3.2 Braided Parallel Indexing vs Data Parallel Partitioned Indexing	9
3.2.1 Massively Parallel Exhaustive Scanning (MPES) on the GPU	11
3.3 Massively Parallel Three-Phase Scanning (MPTS) R-Trees on the GPU	14
3.4 Multi-threaded R-trees Search on the CPU Multi-cores	19
3.5 Evaluation	20
3.5.1 Experimental environment	20
3.5.2 Experimental results	21
3.6 Summary	28
Chapter 4. Exploiting Massive Parallelism of the GPU for Multi-dimensional Indexing	29
4.1 Existing Stackless Ray Traversal Algorithms	29
4.1.1 Exhaustive Scanning on GPU	29
4.1.2 User-defined Stack in Global Memory	30
4.1.3 Kd-restart for Range Query Processing	30
4.1.4 Rope Tree	30
4.1.5 Parent Link	31
4.1.6 Skip Pointer	31
4.1.7 Short Stack for R-tree	32
4.2 Multi-dimensional Indexing Structures on the GPU	32
4.2.1 Massively Parallel Hilbert R-Tree (MPHR-Tree)	33
4.2.2 Multi-way Space Partitioning Bounding Volume Hierarchy	34
4.3 Massively Parallel Tree Traversal Algorithms	34
4.3.1 MPRS: Massively Parallel Restart Scanning	34

4.3.2	Analysis of Massively Parallel Restart Scanning	39
4.4	Evaluation	40
4.4.1	Experimental environment	40
4.4.2	Experimental results	41
4.5	Summary	49
Chapter 5.	Co-Processing Heterogeneous Parallel Index	50
5.1	Hybrid Tree Structure	50
5.2	Query Co-Processing in Hybrid Tree	53
5.3	Multiple Query Scheduling	55
5.4	Evaluation	56
5.4.1	Experimental Environment	57
5.4.2	GPU Kernel Launch Overhead	57
5.4.3	Various GPU Thread Block & Scan Size	59
5.4.4	Throughput of Batch Query Processing	63
5.5	Summary	64
Chapter 6.	Conclusion	66

List of Tables

4.1	Construction Time of Massively Parallel Hilbert R-trees on Tesla M2090	41
5.1	GPU Kernel Launch Overhead with or without Hilbert Space Filling Curve and K-means Clustering	57

List of Figures

3.1	Braided Parallel Indexing vs Data Parallel Partitioned Indexing	10
3.2	<i>Massively Parallel Exhaustive Scanning on the GPU (MPES)</i>	11
3.3	<i>MPTS R-Tree Search with Sibling Check</i>	15
3.4	Throughput with Varying Number of Blocks and Threads per Block	21
3.5	Effect of the Number of Indexed Data	22
3.6	Braided Parallel Processing vs Data-parallel Partitioned Indexing	23
3.7	Throughput and Number of Visited Nodes High Dimensions	24
3.8	Query Execution Time in High Dimensions	25
3.9	Throughput and Number of Visited Nodes with Varying Selection Ratio	25
3.10	Query Execution Time with Varying Selection Ratio	26
3.11	Performance evaluation on various multi-core architectures	27
3.12	Performance evaluation using San Francisco roadmap data	28
4.1	<i>Query Processing with Skip Pointer</i>	32
4.2	<i>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.</i>	33
4.3	<i>Massively Parallel Restart Scanning with MPHR-tree Structure</i>	36
4.4	<i>Query Processing Performance With Braided Parallel Index and Partitioned Index</i>	42
4.5	<i>Search Performance with Varying the Degree of Tree Nodes</i>	43
4.6	<i>Average Query Response Time with MPHR-Tree</i>	45
4.7	<i>Average Query Response Time with disjoint MSP BVH</i>	45
4.8	<i>R-Tree: Search Performance with Varying Selection Ratio</i>	46
4.9	<i>Search Performance with Varying Number of Dimensions</i>	47
5.1	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)	51
5.2	Multiple Query Scheduling	55
5.3	Amounts of Host and Device Memory Access with Varying Scan Size	58
5.4	Query Execution Time of Various GPU Thread Blocks and Scan Size	58
5.5	Speed Up with Various Selection Ratio and Scan Size(Normalized to 128)	59
5.6	Profiled Results with Various Selection Ratio	60

5.7	Performance Results with Various Selection Ratio	61
5.8	Static vs. Dynamic Query Scheduling	63

Chapter 1. Introduction

In this dissertation, we investigate the problem of parallel multi-dimensional range query processing on the GPU. Graphics processing unit (GPU) is now widely used for high-performance parallel computation as a cost effective solution [7]. 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, scientific computing, and computational chemistry, have been successfully accelerated by GPUs [2, 8, 3, 9, 10, 11, 12, 13]. While GPU-accelerated systems achieve superior performance for computationally expensive scientific applications, there still exist many scientific computing domains that have not yet leveraged the parallel computing power of the GPU.

In many scientific disciplines, sensor devices and simulators generate a truly large amount of data. Such datasets are commonly comprised of sets of multi-dimensional arrays where each array has spatial or temporal coordinates; for instance, location and time information from sensor devices in two or three-dimensional spaces. To efficiently navigate such datasets, many scientific data file formats, such as NetCDF [14] and Hierarchical Data Format(HDF) [15], have been developed. 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 [16, 17]. However, such data formats and indexing libraries still do not fully support multi-dimensional indexing that allows direct access to the subsets of datasets using ranges of spatial and temporal coordinates.

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. In 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 [18, 19]. The database community also has improved multi-dimensional indexing structures for the past couple of decades. R-Tree [1] and its variants are most commonly used for multi-dimensional range query processing as of today [20, 21, 22, 23]. 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, 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 [24]. 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 [25, 26].

In computer graphics community, various techniques have been proposed to overcome the problems [27, 28, 29, 30]. Foley et al. [27] proposed to restart tree traversal instead of backtracking. Hapala et al. [28] proposed to enable backtracking via auxiliary *parent link* pointers. Horn et al. [31] 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 they are proven to improve query processing throughput, they are not designed to improve 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 scientific applications domain.

In this dissertation, we present GPU-based stackless tree traversal algorithms - *Massively Parallel Three-phase Scanning (MPTS)*, *Massively Parallel Restart Scanning (MPRS)* and a novel variant of R-trees - *MPHR-tree* [6, 24], which is designed for MPRS traversal algorithm. MPTS and MPRS are alternative tree traversal algorithms that scan 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, brute-force scanning methods on the GPU and other stackless tree traversal algorithms in terms of both query response time and query processing throughput.

We also present *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. 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, this work proposes to scan a large number of leaf nodes in parallel according to the selection ratio of the range query. For such parallel scanning, the GPU is known to be superior to the CPU. The experimental results show that our multi-dimensional range query co-processing scheme improves the query response time by up to 12x and query processing throughput by up to 4x compared to the state-of-the-art GPU tree traversal algorithm.

1.1 Thesis and Contributions

In this dissertation, I support the following thesis statement:

Graphics processing units can improve multi-dimensional range query performance by an order of magnitude or more.

To this end, I design, implement, and evaluate the novel tree structures and tree traversal algorithms to leverage the massive parallelism of the GPU.

The key contributions of this dissertation are summarized as follows.

- **MPTS tree traversal algorithm and the performance comparison with CPU-based R-tree tree traversal algorithm**

The MPRS improves the utilization of GPU architecture for range query processing and avoids the irregular search path by transforming the tree traversal problem into a sequential data processing problem. Our experimental results demonstrate how the MPTS R-Trees algorithm effectively prunes out irrelevant tree nodes while it places very little overhead on the GPU. The search time of MPTS algorithm on the GPU Fermi M2090 is as low as 20% of parallel R-trees on quad-core Intel Xeon E5506 architecture and consistently outperforms brute-force scanning methods.

We have also compared the braided parallel indexing and data-parallel partitioned indexing, and presented experimental results that show braided parallel indexing improves system throughput when a large number of concurrent queries are submitted and data-parallel partitioned indexing helps improve individual query response time. We postulate the two parallel indexing schemes can be adaptively employed in the case when the query arrival distribution changes dynamically.

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

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 (SMP) 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* [32] 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 manner. MPRS avoids visiting the nodes that have visited 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 [31] and parent link [28] 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). To compare their performance with our range query algorithms, 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.

- **Query co-processing on heterogeneous architectures**

We present a novel co-processing scheme for multi-dimensional range query. Our 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.

1.2 Thesis Organization

The remainder of this dissertation is organized as follows. In chapter 2, we describe previous related work. In chapter 3, we briefly introduce CUDA programming model and parallelism for query processing on the GPU. We also present a novel range query algorithm, MPTS, for GPU architecture. Then we compare the indexing performance with CPU-based R-Tree Search Algorithm. Chapter 4, we present our adaptation of stackless tree traversal algorithms and introduce 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 and evaluate and analysis the performance. In Chapter 5, we propose and evaluate our heterogeneous co-processing scheme for multi-dimensional indexing with the-state-of-the-art GPU-based tree structures and tree traversal algorithms. Chapter 6, we conclude this dissertation.

Chapter 2. Related Work

GPU has been repeatedly reported that it offers unprecedented performance in various applications and it has also been studied to improve the performance of database SQL processing in addition to the index parallelization [33, 34, 35, 36, 37]. Bakkum et al. have implemented SQLite database virtual machine on the GPU and improved the performance of SELECT queries [35]. They transformed the internal B+-tree implementation of the database table of SQLite into a straightforward row-column format in order to accelerate the query processing on the GPU. Che et al. [33] reimplemented a set of computationally demanding general purpose applications on the GPU and showed that they can benefit from data parallelism. He et al. implemented a set of data-parallel relational query processing primitives such as map, split, sort, and one-dimensional cache-conscious search tree on GPU [34]. Govindaraju et al. [36] also proposed GPU-aware algorithms for several common database operations such as conjunctive selections, aggregations, and semi-linear queries. However, they did not parallelize the tree traversal algorithms.

In spatio-temporal database community, there has been extensive research on multi-dimensional indexing tree structures, starting with the seminal work on R-trees [1]. R-tree is a balanced tree structure whose tree node consists of an array of minimum bounding boxes (MBBs). The MBB of a tree node is the smallest multi-dimensional box that encompasses all the data in the subtree, i.e., the MBB in R-tree leaf node encloses nearby spatial objects and the MBB of internal tree nodes encloses all the underlying MBBs of lower level sub-trees in a hierarchical way. There were also some efforts to parallelize the R-trees in shared-nothing environment [38, 39]. Kamel proposed Multiplexed R-trees [40], Koudas et al. proposed Master R-trees [41], for a machine with a single CPU and multiple disks. For distributed parallel cluster machines, Master Client R-trees was proposed by Schnitzer et al. [42]. Nam et al. compared the challenges and problems of designing distributed multi-dimensional indexes for data-intensive scientific applications [39]. The distributed multi-dimensional indexing structures have numerous usages in various contexts including distributed and parallel query processing systems [43].

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 SMPs have been minimally enlarged [44]. CUDA threads use the fast shared memory as their run-time stack, but the latest Tesla GPUs have only 64K bytes of shared memory. Due to the tiny stack sizes, recursive search algorithms of multi-dimensional indexing structures often fail. In addition, it is well known that hierarchical tree structures are inherently not well suited for parallel processing because their irregular memory access patterns make it difficult to exploit a large number of processing units

on the GPU. 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.

For data parallel tree-structured index, Zhou et al. proposed to compare multiple keys of B+-trees at the same time using SIMD instruction [45]. Kaldewey et al. [46] also proposed a parallel search algorithm, called *P-ary search*, for one-dimensional sorted lists and showed that it outperforms binary search algorithm on GPU. Kim et al. presented *FAST (Fast Architecture Sensitive Tree)*, which rearranges a binary search tree into tree-structured blocks to maximize data-level and thread-level parallelism on GPU architecture [47]. Each block of FAST is the unit of parallel processing in a single streaming multiprocessor (SMP) of GPU. The block of FAST is similar to the node of disk-based R-trees in a sense that its size is chosen to avoid the bandwidth bottleneck between main memory and GPU device memory. These works are one-dimensional data structures that do not need back-tracking. For multi-dimensional range queries, there can be several child nodes to visit. After one path is taken and the path search is completed, it is necessary to back-track to the last place where there were multiple choices in paths so that another path can be taken. Since recursive back-tracking algorithm does not perform well in current GPU architecture, we propose a sequential range query processing algorithms.

To avoid the recursive tree traversal in multi-dimensional indexing, Luo et al. [48] proposed a parallel R-tree traversal algorithm on the GPU. They employ a queue in the shared memory of SMP to avoid recursion(i.e., stack operations). Their algorithm transforms the R-tree search into a breadth-first search (BFS). But storing tree nodes to visit in the shared memory is not very scalable since GPUs provide very small shared memory space. Moreover, the shared queue requires atomic write operation which hurts concurrency and performance. Our studies are more scalable since it does not depend on the size of shared memory.

In the computer graphics community, various techniques have been proposed to overcome the tree recursion problem on the GPU [27, 28, 29, 30, 49, 50]. Foley et al. [27] proposed *kd-restart* algorithm for ray tracing using kd-trees. Later, Hapala et al. [28] 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 an auxiliary *parent link* pointer to its parent node so that it can backtrack to the parent node by following the pointer. Horn et al. [31] 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 but fixed-sized stack leaks from the bottom when it becomes full. While these algorithms are designed for computer graphics applications and 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 that of the number of rays in computer graphics. Hence, such task-parallel stackless tree traversal algorithms are not sufficient in the scientific applications domain. Although kd-restart algorithm can not be employed for multi-dimensional range query processing because it tracks crossing points of the ray with the region boundaries of kd-tree leaf nodes, the stackless tree traversal algorithms proposed by Hapala et al. [28], Horn et al. [31], 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 Chapter 3.

Recently, Shahvarani et al. [51] proposed the HB+tree, similar to our Hybrid tree, 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. The 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 the one-dimensional query, whereas we propose a co-processing scheme for multi-dimensional range query in Chapter 4.

Nearest neighbor query (k-NN) is another important class of multi-dimensional query that finds closest points given a set of points in multi-dimensional space [52, 53, 25]. Garcia et al. [54] have proposed brute-force k-NN search algorithms for the GPU and Cayton presented a two-level brute-force search algorithm called *Random Ball Cover (RBC)* [55]. In RBC algorithm, some random points are used as representative points for subsets of the dataset, and the amount of work for nearest neighbor queries are substantially reduced by pruning out some subsets using the representative points. This work is different from ours in that RBC targets high dimensional point datasets, while our work focuses on the relatively low dimensional range queries and proposes a novel tree traversing algorithm for hierarchical tree structures are known to be very effective for low dimensional datasets.

FastQuery is a parallel indexing software framework designed for modern supercomputing platforms [17]. It exploits parallelism on distributed multi-core resources using one-dimensional bitmap indexing. It builds a huge bitmap index for sub-arrays for large scientific datasets and processes queries in parallel by partitioning the bitmap index. For multi-dimensional scientific queries, Su et al. [56] proposed a parallel multi-level bitmap indexing scheme that supports partitioning queries over multi-dimensions based on query profile. Our work is different from theirs in a sense that our tree traversal algorithms leverage a large number of cores in GPU accelerators while their indexing framework targets cluster servers and it does not employ GPUs. Our experimental results show the MPHR-trees and Hybrid Tree can be easily extended to large scale GPU cluster servers.

Chapter 3. Multi-dimensional Range Query Processing with R-Tree on the GPU

In this chapter, we propose a novel *MPTS (Massively Parallel Three-phase Scanning)* R-tree traversal algorithm for multi-dimensional range query, which converts recursive access to tree nodes into a sequential access. Our extensive experimental study shows that MPTS R-tree traversal algorithm on NVIDIA Tesla M2090 GPU consistently outperforms traditional recursive R-trees search algorithm on Intel Xeon E5506 processors.

3.1 Background on CUDA

To process a large amount of data in parallel, a CUDA program spawns thousands of extremely lightweight parallel threads, and they execute the *kernel function* on the GPU to access small portions of the large input dataset in parallel. A CUDA *thread block* consists of a set of CUDA threads that share intermediate data results and cooperate on memory access with the other threads through synchronization mechanisms that CUDA provides. The threads in the same thread block can efficiently share data through a small but low latency *shared memory*. In addition to the small shared memory, GPU comes with a relatively large DRAM memory, called *global (device) memory*, which is shared and accessed by all CUDA threads.

A *warp* is the minimum thread scheduling unit in CUDA architecture, but the warp is not controllable by programmers. Instead, programmers can specify the number of *blocks* and the number of *threads* per block when invoking a CUDA kernel function. The blocks are distributed across the multiple SMPs, and multiple threads in a single block are executed by a set of CUDA processing units in a single SMP concurrently. Tesla K20m GPU contains 192 CUDA processing units, and it can execute a single warp of 32 threads in a single clock cycle on average. Note that the number of concurrent threads in a block can be limited if the amount of memory (CUDA registers, shared memory, and constant memories) required by threads exceeds the capacity of memory that reside in the SMPs.

3.2 Braided Parallel Indexing vs Data Parallel Partitioned Indexing

In GPU computing, *braided parallelism* implies that multiple independent jobs run in parallel on different SMPs and each independent job is processed in a data parallel fashion across multiple processing units in a single SMP. Braided parallelism is commonly used in GPU ap-

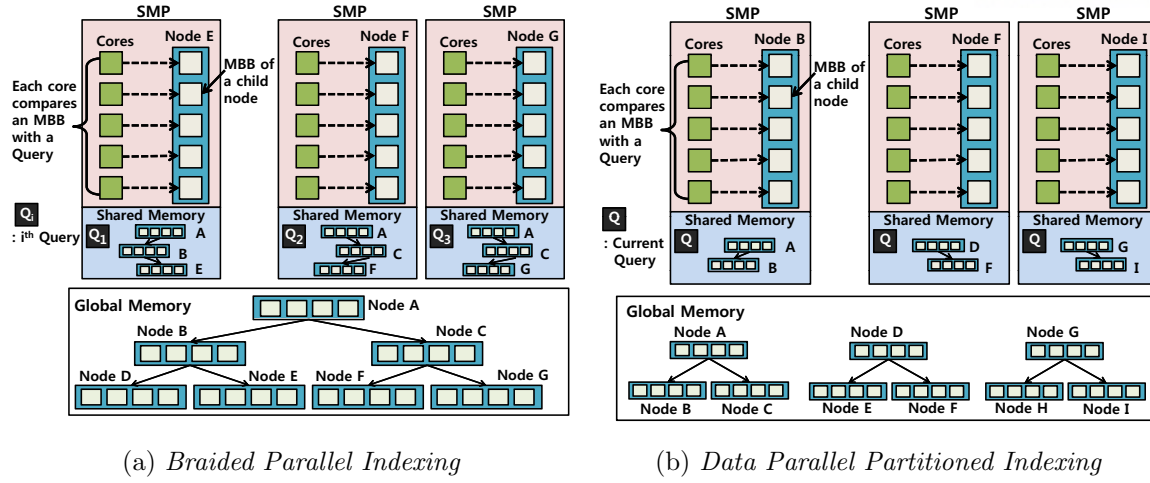


figure 3.1: Braided Parallel Indexing vs Data Parallel Partitioned Indexing

plications since it fits nicely with SIMD architecture of the GPU. However, braided parallelism does not scale when the number of submitted tasks is small.

Since maximizing the utilization of GPU processing units plays the key role in improving the performance of GPU applications, we compared two approaches that parallelize index search operations. One method is braided parallelism that assigns a different query to each SMP, i.e. a GPU that has 16 SMPs can execute 16 queries concurrently. Since there's only a single index in GPU memory, the index will be shared by all the SMPs, but different parts of the index will be accessed to serve different queries. As task parallelism scales with a large number of concurrent jobs, this braided parallel query processing improves query processing throughput when a large number of queries are continuously submitted. However, it would not help reduce the execution time of running each query.

In order to improve the response time of the individual query, we devised another method that makes maximum use of *data parallelism*, where we partition the index into sub-indexes and distribute them to each SMP. Partitioning spreads and decreases the amount of work to be done for a single query across multiple SMPs because each SMP has a smaller partitioned index to work on. Partitioning a large index will decrease the size of the index by a factor of the number of SMPs. When a range query is submitted, all the SMPs compare the same given query range with its own partitioned index and returns the list of the data objects whose multi-dimensional coordinates overlap with the given range. This approach can help decreasing the query execution time of a single query since the amount of work is reduced and spread across more number of processing units.

Figure 3.1 illustrates the differences between braided parallel indexing scheme and data parallel partitioned indexing scheme. To refer to the data parallel partitioned indexing, we use the term partitioned indexing for short. In braided parallel indexing shown in Figure 3.1a, each

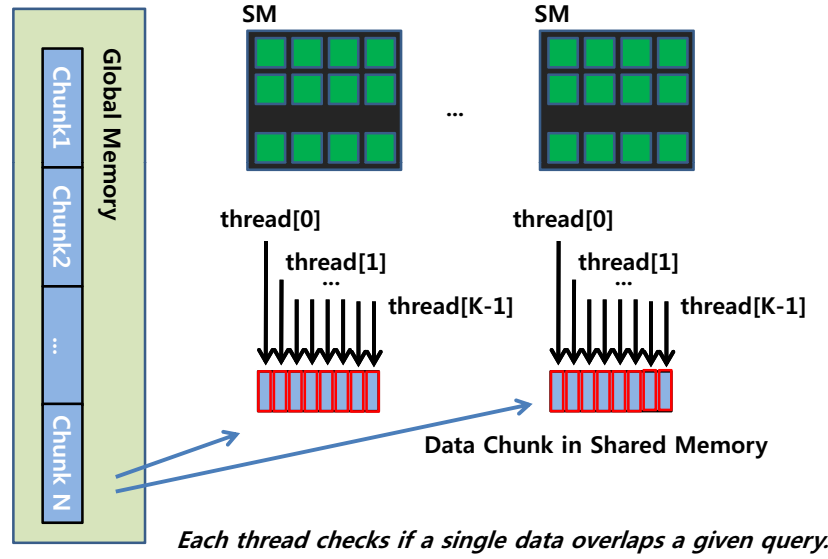


figure 3.2: *Massively Parallel Exhaustive Scanning on the GPU (MPES)*

SMP processes different user query, hence if the fewer number of queries are submitted than the number of available SMPs, the utilization of processing units would be poor. However, in data parallel partitioned indexing shown in Figure 3.1b, the same single query is processed by all SMPs concurrently with different partitioned indexes, thus utilization would be higher than that of braided parallel indexing even when the number of submitted queries is small.

3.2.1 Massively Parallel Exhaustive Scanning (MPES) on the GPU

Although GPU programming model such as CUDA or OpenCL has improved for general purpose applications, GPUs are still very restrictive in many senses. For examples, SIMD execution model of GPUs is not well suitable for irregular data access patterns because branching is very time-consuming operation on the GPU. If threads of a warp take different branch paths due to data-dependent conditions, the warp will execute each thread serially. In order to avoid this problem, data-dependent algorithms should be carefully redesigned. An algorithm of traversing a tree-structured multi-dimensional index is one of the data-dependent algorithms as we will describe in section 3.3.

An easy way of taking the advantages of a large number of processing units on GPU is to apply brute-force algorithms. In various fields including multi-dimensional query processing, brute-force algorithms on GPUs are drawing attentions since it is completely data-independent and effectively utilizes a large number of processing units. In database community, brute-force search algorithms for high dimensional datasets have been extensively studied in the literature because of its superior search performance to other sophisticated tree-structured indexes when datasets are in high dimensions [57]. The traversal algorithms of R-Trees and its variants are

Algorithm 1 *R-tree Search Algorithm*

```

void RTreeSearch(Node* n, MBB* query)
1: // this is an internal node in the tree
2: if  $n.level > 0$  then
3:   for  $i \leftarrow 0, Number\_Of\_Child\_Nodes$  do
4:     if  $MBBOverlap(query, n.child\_mbb[i])$  then
5:        $RTreeSearch(n.child[i], query);$ 
6:     end if
7:   end for
8: else
9:   // this is a leaf node
10:  for  $i \leftarrow 0, Number\_Of\_Child\_Nodes$  do
11:    if  $MBBOverlap(query, n.child\_mbb[i])$  then
12:      // Found overlapping data
13:       $SaveOverlappingData(n.child[i]);$ 
14:    end if
15:  end for
16: end if

```

designed to visit as small number of tree nodes as possible, i.e. in log-scale. Thereby recursive search functions, as shown in Algorithm 1, or using while-loop with user-defined stack or queue are being used for the irregular tree node traversal patterns. The recursive search algorithms that prune out unnecessary nodes have been shown to outperform brute-force search algorithms for low dimensional datasets in many studies. However, for high dimensional datasets, the recursive search algorithms suffer from the well known *curse of dimensionality* problem - the exponential growth of hyper-volume as a function of dimension [58]. For high dimensional vector datasets, k-nearest neighbor (kNN) queries [59] are more common access patterns than orthogonal range queries, and it has been shown that the brute-force kNN search algorithms on GPU show good performance [55, 54].

One of the features that distinguishes CUDA programming model with CPU based programming model is its limited support of recursion. In CUDA programs, recursive algorithms can cause memory problems because very small and slow off-chip local memory is used for runtime stack. For instance, a Tesla M2090 GPU has only 48KB L1 cache and 512K off-chip local memory. For this reason, early CUDA programming model did not support recursion at all. Although CUDA started supporting recursive function calls since version 3.1, the recursive functions can easily crash if the size of function arguments is large. In CUDA 5.0, dynamic parallelism allows a kernel function to call other kernel functions recursively, but still it does

Algorithm 2 *Braided Parallel MPES Algorithm on the GPU*

void BraidedParallelMPES(MBB* node, MBB* query, int *totalHit, int numThreads, int numData)

```

1: // GPU block id and thread id
2: int bid ← blockIdx.x;
3: int tid ← threadIdx.x;
4: for i ← 0, numData do
5:   if node[i + tid].boundary.contain(query[bid].boundary) then
6:     saveResult(tid, node[i + tid].data);
7:   end if
8:   i += numThreads;
9: end for

```

Algorithm 3 *Data-Parallel Partitioned MPES on the GPU*

void DataParallelPartitionedMPESonGPU(MBB* node, MBB* query, int *totalHit, int numThreads, int numBlocks, int numData)

```

1: // GPU block id and thread id
2: int bid ← blockIdx.x;
3: int tid ← threadIdx.x;
4: int start ← (numData/numBlocks) * bid;
5: int finish ← (numData/numBlocks) * (bid + 1);
6: for i ← start, finish do
7:   if node[i + tid].boundary.contain(query.boundary) then
8:     saveResult(tid, node[i + tid].data);
9:   end if
10:  i += numThreads;
11: end for

```

not provide a solution to the tiny size of run-time stack in GPU architectures. Instead of using the recursive functions, user-defined stack in global memory can be employed, but at the cost of significant performance degradation [60]. Therefore, if possible, it is desirable to modify and redesign recursive algorithms and transform random data access patterns into sequential data processing in order to maximize the utilization of a large number of GPU processing units and get the maximum performance out of SIMD architecture.

In section 3.3, we describe how the recursive tree traversal algorithm that needs backtracking has been transformed into a sequential range query algorithm that effectively eliminates memory space problems and prunes out unnecessary nodes. In order to evaluate the performance improvement over the brute-force algorithms, we implemented a *massively parallel exhaustive search (MPES)* function as a performance baseline as shown in Algorithm 2 and 3. MPES

search function simply transforms a range query search operation into a stream data filtering process that is well suited for GPU acceleration. As illustrated in Figure 3.2, MPES divides the total number of multi-dimensional data objects - N by the number of CUDA threads in each SMP - K , and each thread compares the N/K number of data elements with a given query range to detect whether they overlap or not. For the data-parallel partitioned indexing approach described in section 3.2, each thread compares the $N/(K \cdot M)$ number of data elements where M is the number of SMPs.

3.3 Massively Parallel Three-Phase Scanning (MPTS) R-Trees on the GPU

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.

In order to address this problem, we propose and discuss an alternative tree traversal algorithm that navigates tree nodes sequentially in this section, which we named *Massively Parallel Three-phase Scanning (MPTS)*. This algorithm is very simple but as effective as recursive search function as we will show in the experimental section. As shown in Algorithm 1, traversing R-Trees on CPU needs a loop to iterate the array of minimum bounding boxes (MBBs) of child tree nodes. To exploit massively parallel CUDA architecture and to alleviate the I/O resource competition, this loop is parallelized with a number of threads. In MPTS search algorithm, a set of threads in a CUDA block cooperate to check in parallel if the MBBs of child nodes overlap a given query, i.e., the number of threads in each block is set equal to the number of node fan-outs (the maximum number of child nodes). Since current GPU architecture executes an instruction in the unit of warp (32 threads), the number of node fan-outs should be equal to a multiple of 32.

This parallel search scheme is desirable for SIMD architecture since all the threads in a single block read the same tree node and each thread independently determines whether a child

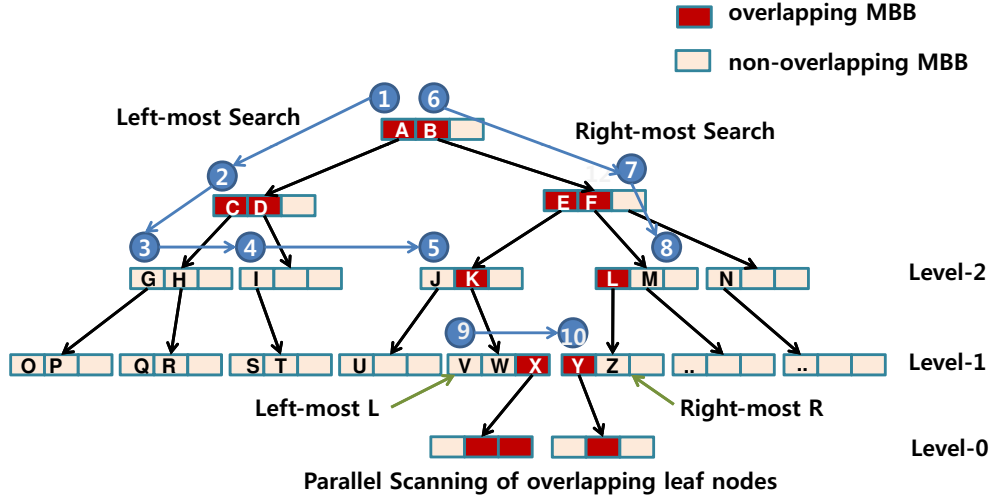


figure 3.3: MPTS R-Tree Search with Sibling Check

node overlaps a given range query. After all the threads are done with comparing a query with MBBs of child nodes, they should agree with which child node to visit next if there are more than one overlapping child node. The recursive search algorithm shown in Algorithm 1 navigates down one of the overlapping nodes, and it back-tracks to the current node so that it visits another overlapping node. This recursion needs a large run-time stack space especially when the size of the tree structure is large. Current run-time stack frame stores which child nodes of the current node overlap so that when it back-tracks to current stack frame it restores the overlap information without comparing the MBBs and chooses the next child node to visit. However, the recursive range query function is not scalable since it often fails when the size of the index is large and query range is also large.¹

To avoid back-tracking, MPTS search algorithm selects at most one child node to visit no matter how many child nodes overlap a query. As shown in Figure 3.3, MPTS search algorithm keeps choosing the leftmost node in each level in the first phase, (Steps 1-3), and in the second phase, the rightmost node in each level is visited (Steps 4-6). Any node that is not in between the leftmost and rightmost nodes has no chance of overlapping the query. If there is an overlapping node out of the leftmost and rightmost nodes, it contradicts that they are the leftmost or rightmost nodes. This pruning process determines which nodes are irrelevant and reduces the number of tree nodes to visit. The leftmost and rightmost scanning algorithm is

¹In order to overcome this drawback, we implemented a non-recursive R-tree search function that stores overlap information in global memory. I.e., recursion is converted into a loop where a thread inserts the address of overlapping child nodes into a queue in global memory. When a block of threads are done with processing current tree node, they fetch the next tree node from the queue, compare the MBBs of its child nodes again, and repeat. Unfortunately, it turned out that this design makes the global memory access a serious performance bottleneck and performs extremely poor. Thus we do not show its performance in experimental results.

Algorithm 4 *Leftmost/Rightmost Scanning Algorithm of MPTS*

Ndoe* Find_LeftRightmostNode(Node *root, MBB *query, int LR-
Flag)

```

1: shared int Ovlp[NumberOfChildNodes];
2: shared Node node;
3: node  $\leftarrow$  root;
4: while node  $\neq$  NULL do
5:   if node's level is higher than one then
6:     // internal nodes, keep choosing left/rightmost child
7:     if node.child[tid].child and MBBOverlap(query, node.child[tid].mbb) then
8:       // if child[tid] exists and the MBB of child[tid] overlaps a query
9:       Ovlp[tid]  $\leftarrow$  tid;
10:    end if
11:    syncthreads()
12:    // parallel reduction to find out
13:    // the leftmost/rightmost overlapping child
14:    // Ovlp[0] holds the index of leftmost/rightmost child
15:    Ovlp[0]  $\leftarrow$  parallelReduction(Ovlp, LRFlag)
16:    if none of the branches overlaps then
17:      node = NULL;
18:      if LRFlag == LEFT and there is a right sibling then
19:        node  $\leftarrow$  (Node*)((char*)node + TREENODESIZE)
20:      end if
21:      if LRFlag == RIGHT and there is a left sibling then
22:        node  $\leftarrow$  (Node*)((char*)node - TREENODESIZE)
23:      end if
24:    else
25:      // fetch the leftmost/rightmost child node
26:      node  $\leftarrow$  node.branch[Ovlp[0]].child;
27:    end if
28:    syncthreads()
29:  else
30:    // this is a level-1 node (a parent of the leaf node)
31:    return(long)node;
32:  end if
33: end while

```

given in Algorithm 4. After identifying the leftmost level-1 (parent of leaf node) node and the rightmost level-1 node, all the level-1 nodes in between are scanned in the last phase as shown

in Algorithm 5. If a level-1 node has an MBB of child leaf node that overlaps the query, the child leaf node is fetched and compared against the query.

In the example shown in Figure 3.3, let us assume a single warp consists of three threads and the maximum child nodes of each tree node are also three. In step one (circled one in the figure), two threads will find the red-colored left and middle MBBs (A and B) of the root node overlap a given query range. The third thread will find out that the root node doesn't have third child node and wait for the other two threads to finish. In the leftmost search phase, the middle overlapping MBB B will be ignored but the left child node A will be chosen and visited. A simple parallel reduction algorithm can be employed to identify which overlapping MBB is located in the leftmost position in the tree structure. In order to avoid shared memory bank conflicts, we employed sequential addressing for the parallel reduction. In step 2, again the first and second threads find out the left and middle MBBs (C and D) overlap, and C will be chosen just because it is located in the leftmost position among them. In step 3, threads will find out none of the MBBs (G and H) overlap. In traditional recursive tree traversal algorithms, we should go back to the parent node, but back-tracking should be avoided in GPU environment. Instead, we can blindly navigate down further following the rightmost child node (i.e. $H \rightarrow Q$ and R) although we know they do not overlap. This approach will increase the distance between leftmost and rightmost nodes and the probability of false hits. Although it will hurt the query processing performance as a result, it does not harm correctness of the query results since the real leftmost (rightmost) overlapping node will be located on the right (left) side of the false leftmost (rightmost) nodes. In the last parallel scanning phase, the nodes between leftmost and rightmost nodes are scanned in parallel and any non-overlapping nodes are filtered out.

A better way of avoiding false hits and reducing the distance between leftmost and rightmost nodes is the *sibling jump* shown in steps 3, 4, and 5 of Figure 3.3, which fetches its right sibling node when there's no overlapping MBB in the current node. In our implementation of MPTS search, we rearrange tree nodes in a breadth-first manner while transferring the index from host memory to GPU global memory, thus it is trivial to fetch an adjacent sibling. The tree index in GPU global memory is stored in a single contiguous block, thus we can easily calculate the memory address of adjacent siblings by adding the fixed tree node size to the current node's memory address. In step 3 shown in Figure 3.3, instead of visiting the child node that has Q and R , MPTS jumps to its sibling node in step 4. Again the MBB I of the sibling node also does not overlap, hence it jumps again to the node that has J and K . When none of the MBBs overlap, MPTS keeps jumping until the sibling node has an overlapping child node or there is no more sibling node. In step 5, J does not overlap but K does, and the child node pointed by K is the level-1 node. So we return the memory address of the child node as the leftmost level-1 node. Note that as we jump to siblings in the higher level of trees, the number of pruned out

Algorithm 5 *Braided Parallel MPTS R-Trees Algorithm*

```

void MPTS_RTree_RangeQuery(Node* root, MBB* query)
1: // GPU block id and thread id
2: int bid ← blockIdx.x;
3: int tid ← threadIdx.x;
4: // search leftmost and rightmost overlapping node in the level one
5: Node* leftMost ← Find_LeftRightmostNode(root, query[bid], LEFT);
6: Node* rightMost ← Find_LeftRightmostNode(root, query[bid], RIGHT);
7: if leftMost == NULL or rightMost == NULL then
8:     return;
9: end if
10: while leftMost ≤ rightMost do
11:     // fetch the next level one node and filter it out.
12:     for i ← 0, leftMost.NumOfChilds do
13:         Node* leaf ← leftMost.child[i];
14:         if MBBOverlap(query[bid], leaf.child[tid].mbb) then
15:             saveResult(tid, leaf.child[tid].data);
16:         end if
17:     end for
18:     leftMost += TREENODESIZE;
19: end while

```

leaf nodes increases exponentially.

In the second phase, MPTS search algorithm will find out the rightmost level-1 node in a similar way. In the last parallel scanning phase, the level-1 nodes between the leftmost and the rightmost nodes are scanned and when the MBB of their leaf nodes overlap the query, the leaf node is accessed and the overlapping data object will be returned (step 9 and 10). The leftmost search and the rightmost search phases can run concurrently in a separate group of threads for further optimization. But the overhead of the first and second phase is not very significant since the number of tree nodes that need to be fetched from global memory is just $O(\log_k N)$, where the N is the number of indexed data.

As we navigate down the trees we access the one and only child node in each level. Hence MPTS search algorithm does not require back-tracking or global memory access. The penalty of eliminating the back-tracking is that we may have to visit more number of leaf nodes. The leftmost leaf and the rightmost leaf node can be located very far from each other in the tree structure. In traditional R-trees, the MBBs of a tree node are stored in random order. Thus, MPTS search algorithm might have to scan all the leaf nodes even for a very small query range. Although CUDA is known for its outstanding performance of processing consecutive

data in parallel, scanning all the leaf nodes should be avoided for performance reasons. The performance of MPTS search algorithm depends on how many tree nodes are accessed. Usual tree height of R-trees is 4 or 5 when the tree node size is 4K bytes and the number of the indexed data object is about a million. Thus, leftmost and rightmost search phases are not significant overhead, but the number of visited nodes in the last scanning phase can be $O(n)$ in the worst case where n is the number of indexed data. In order to narrow down the distance between the leftmost leaf node and rightmost leaf node, we employed *Hilbert space filling curve* [61] which is well known for its property of preserving spatial locality. Using Hilbert space filling curve, we rearranged MBBs of an R-tree nodes so that sibling nodes have very high spatial locality. When tree structure has higher spatial locality between tree nodes, the range of search paths becomes narrower and the number of leaf nodes between the leftmost leaf node and the rightmost leaf node decreases.

3.4 Multi-threaded R-trees Search on the CPU Multi-cores

Multi-core processors are now being widely used in many application domains including scientific applications. But the performance of the applications is mainly affected by their algorithms, i.e., how much portion of the algorithms is parallelized. The MPTS search algorithm is not well suited for multi-core processors for many reasons. First of all, the performance of modern CPU cores is much faster than GPU processing units. CPUs do not have problems with conditional branch and recursion. Also, the number of cores in multi-core processors is not as large as the preferred number of tree node fan-outs. Moreover, L1 caches and local memory each core of multi-core processors has will be wasted by redundant data.

Hence, we implemented two simple parallel R-tree indexing schemes on multi-core architectures in a different way from MPTS. One way is to let all the cores share the same R-tree index and assign a different query to each core. We will refer to this scheme as *RTree-MultiCore* in the experimental section. The other parallel indexing scheme is to partition a single R-tree into the number of cores in a multi-processor, and a single query is processed by all the cores that search their own partitioned small indexes. We denote this indexing scheme as *partRTree-Multicore* (*Partitioned R-trees Search on CPU Multi-cores*). This partRTree-Multicore parallel indexing parallelizes a single search operation of R-trees while RTree-Multicore does not. Note that in partRTree-Multicore the number of partitioned-trees (N) is dependent on the number of available cores in CPU, but the N can be set greater than the number of cores to improve resource utilization.

This partRTree-Multicore approach allows us to search small independent indexes in parallel. When inserting a new data, we choose an index to store the data in a round-robin fashion.

Alternatively, we can employ *Hilbert space filling curve* [61] as a de-clustering method so that each core gets a similar amount of load for any shape of range query and we can maximize parallelism. However, in our experimental study not shown in the chapter, the simple round-robin assignment performed almost equally well with a Hilbert space filling curve. The experimental results presented in section 3.5 used round-robin selection.

3.5 Evaluation

3.5.1 Experimental environment

We measure the search performance of the proposed parallel range query algorithms on a machine running Centos 5.8 with CUDA Toolkit 4.1. The machine has Intel Xeon 8 Core E5506 2.13GHz processor, 12GB DDR3 memory, and *Tesla Fermi M2090* GPU card. The Tesla Fermi M2090 has 16 SMPs and each SMP has 32 GPU processing units, which enables 512 (16x32) threads to run concurrently.

To evaluate the proposed parallel indexing schemes, we used the *spatial data generator* [62] developed by Yannis Theodoridis and generated synthetic 4 dimensional point data sets in uniform, normal, and Zipf's distribution, but we only present the experimental results of uniform distribution since the results of the other distributions are similar. We also evaluate the performance of parallel algorithms using real datasets - two dimensional MBBs that encompass California polyline streets.

As for the performance metrics, we measure *TBQET* (*Total Batch Query Execution Time*) for query processing throughput and *QET* (*Query Execution Time*) for query response time. TBQET is the elapsed time between the time when the first query in a batch was submitted and the time when the last query in the batch was finished, i.e. small TBQET implies high system throughput. The other metric, *QET* is the elapsed time between the time when a single query was submitted to GPU kernel function and the time when the query was returned back to the host. We average the QET for 1000 queries, which shows how fast an individual query is executed and returned.

As an optimization effort, memory coalescing is commonly used in CUDA programming in order to reduce the number of memory transactions. As an R-tree node consecutively stores MBBs for CUDA threads, each thread accesses adjacent memory space for each child node. Thus memory coalescing optimization can be naturally dealt with in R-trees. Another commonly used ad-hoc optimization technique that can be employed for further performance improvement is data transfer overlapping with kernel execution so that a new query and previous query results can be transferred while searching R-trees. In the experiments shown below, we did not employ such ad-hoc optimizations for both the GPU and CPU. With such ad-hoc optimizations,

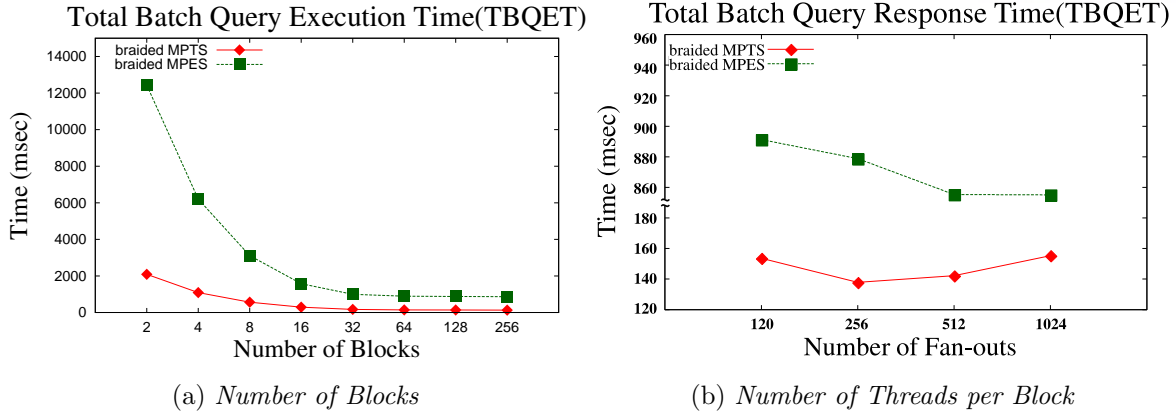


figure 3.4: Throughput with Varying Number of Blocks and Threads per Block

including SSE (Streaming SIMD Extensions), the novel R-tree traversal algorithm and the traditional recursive R-tree search would perform faster than presented in the following section. We compiled the codes with default optimization options using nvcc 4.1 and gcc 4.1.2.

3.5.2 Experimental results

We run experiments with various numbers of blocks and threads to find out when the utilization of GPU processing units is maximized while avoiding resource contention. Figure 3.4 shows the TBQET for 1000 4 dimensional queries. As we increase the number of blocks up to 128, the elapsed search time decreases sub-linearly as illustrated in Figure 3.4a. When the number of blocks is greater than 128, the search time improvement seems minimal or sometimes we observe that the performance gets worse.

For the experiments shown in Figure 3.4b, we increase the number of threads. Note that the number of threads in each block is equal to the number of node fan-outs for the MPTS R-trees as we discussed in section 3.3. The search performance of MPES improves as we increase the number of threads up to 512, however, a larger number of threads don't help improve the utilization of GPU processing units and it does not reduce the search time either. On the contrary, MPTS R-trees shows the best performance when the number of fan-outs (threads) is 256. When the number of fan-outs is smaller than 256, the search performance is slow because the utilization of GPU processing units is low. As the number of fan-outs increases, the tree height of R-trees decreases but the amount of work to be done by a single block increases and it also hurts the search performance because an SMP is flooded with too many threads. If the number of fan-outs decreases, the tree height of MPTS R-trees increases but less amount of work is assigned to a single block, which hurts the utilization of GPU processing units. The best search performance of MPTS R-trees is observed when the number of threads is 256, and this number is different from the best number of MPES because the search pattern of MPTS

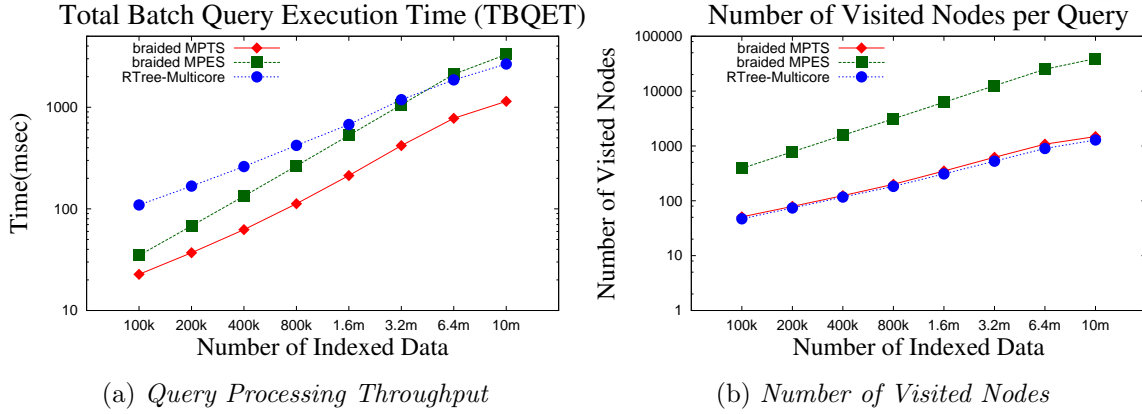


figure 3.5: Effect of the Number of Indexed Data

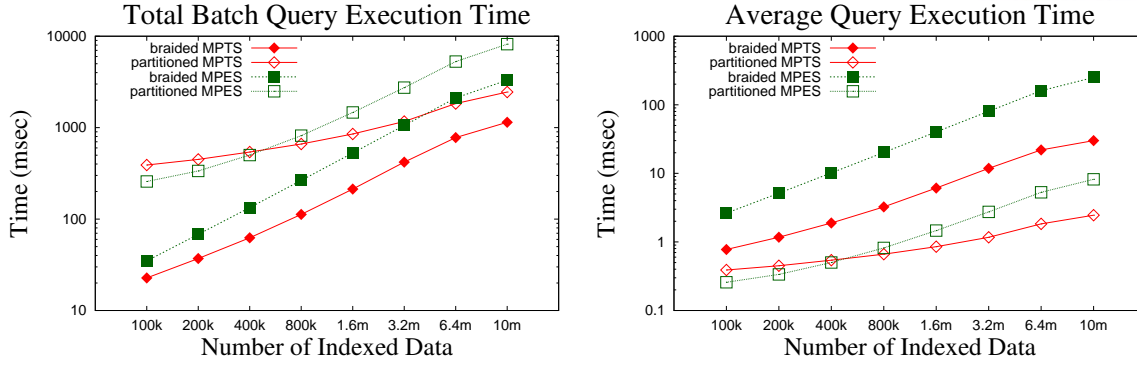
R-trees is different from MPES. For the rest of the experiments, we fix the number of CUDA blocks to 128 and fixed the number of node fan-outs to 256, which is the best configuration for the MPTS R-trees.

In Figure 3.5a, we measure the total batch query execution time (TBQET) for a thousand of 4-dimensional queries as we increase the number of indexed data from 100 thousands to 10 millions. The range query selection ratio² used for this set of experiments is 1%. For MPTS R-trees, the number of blocks is 128 and the number of threads (node fan-outs) is 256. The performance of RTree-Multicore (Multi-Threaded R-trees on CPU) is measured with 4 threads on Intel Xeon E5506 processor.

As the number of indexed data increases, MPES needs to scan more number of data and the search time increases linearly. Note that both the x-axis and y-axis in Figure 3.5a are in log-scale. The search time of MPTS R-tree increases as more number of data are indexed, but the search time of MPES and RTree-Multicore is getting slower at a faster rate as we index more data. The performance gain of MPTS R-trees mostly comes from the less number of visited nodes. Figure 3.5b shows that the number of visited nodes of MPTS R-trees is about 1030 ~ 2465 times less than that of MPES. An interesting result is that the number of visited nodes of MPTS R-trees is almost the same with that of RTree-Multicore. In our experiments, MPTS R-trees visit about 12% (20% in the worst case) more tree nodes than recursive R-trees search algorithm on average. This result is showing that three-phase scanning is almost as effective as recursive tree traversal algorithm and has a great potential to be a replacement of tree traversal algorithms on the GPU.

The next sets of experiments compare the performance of the *data-parallel partitioned indexing* and the *braided parallel indexing* that we discussed in section 3.2. Data-parallel partitioned indexing focuses on improving the response time of a single query while the braided

²Selection ratio refers to the ratio of the number of selected data to the number of indexed data for a query.



(a) *Throughput with varying number of indexed data* (b) *Execution time with varying number of indexed data*

figure 3.6: Braided Parallel Processing vs Data-parallel Partitioned Indexing

parallel batch query processing focuses on improving the overall system throughput when many queries are concurrently submitted to the system.

Figure 3.6a shows that braided parallel query processing with MPTS R-trees outperforms the data-parallel partitioned MPTS R-trees indexing in terms of the system throughput as we expected. However as we index more number of data, the performance gap between braided parallelism and data parallel partitioned indexing becomes smaller, but still significant. With a larger number of indexed data, MPTS R-tree search is much more effective than brute-force MPES, hence even the data-parallel partitioned MPTS R-trees outperforms the braided parallel MPES when the number of index data is larger than 3 millions.

In the experiments shown in Figure 3.6b, we evaluate the average query response time of individual queries. When the number of indexed data is less than 400,000, partitioned MPES yields the fastest QET. However as the number of indexed data grows, the data-parallel partitioned MPTS R-trees outperforms partitioned MPES, which indicates that tree structured indexing and MPTS search algorithm helps reduce the query response time of individual queries especially when the size of the dataset is very large. Although the braided parallel query processing improves the system throughput as shown in Figure 3.6a, we show from this set of experiments that it is not effective in improving query response time.

Figure 3.7 and 3.8 show the performance of index search for various dimensions, from two to sixty-four dimensions. For the experiments in high dimensions, we generate 1 million uniformly distributed hyper-cube data varying the number of dimensions, and submit 1,000 synthetically generated range queries in uniform distribution. The average selection ratio of the queries is adjusted to 1% for all the dimensions.

With high dimensional datasets, the exponential growth of the space causes various phenomena related to the curse of dimensionality problem. As discussed in section 3.2.1, it has

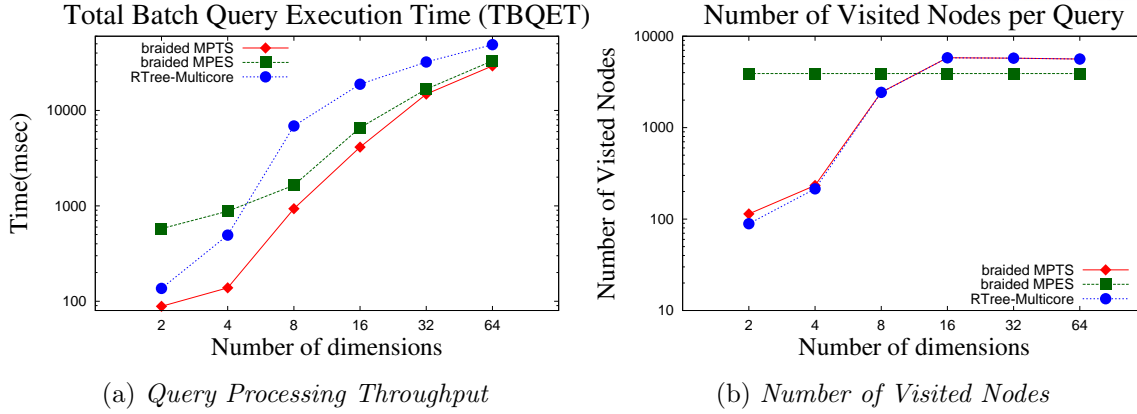


figure 3.7: Throughput and Number of Visited Nodes High Dimensions

been well known that brute-force linear scanning works faster than sophisticatedly designed tree structured indexes when the datasets are in high-dimensional space [23]. Our experiments also confirm this fact. As the dimension increases, multi-threaded R-trees (RTree-Multicore) do not perform well and its search time becomes much slower than that of MPES.

The three-phase scanning algorithm of MPTS R-trees works in a very similar way with brute-force linear scanning except that it reduces the range of scanning by the help of tree-structured indexing. When the dataset is in lower dimension than 16, MPTS R-trees win MPES by big time, but as the dimension increases, the performance gap between MPTS R-trees and MPES keeps decreasing because of the notorious curse of dimensionality problem.

The number of visited nodes shown in Figure 3.7b illustrates how efficiently the three-phase scanning algorithm prunes out non-overlapping leaf data. The three-phase scanning algorithm of MPTS R-trees visits almost the same number of nodes with the regular recursive R-tree search algorithm. In the worst case, about 28% more nodes were visited when the datasets are in 2 dimensions. With 4 dimensional datasets, 8% more nodes were visited. But less than 1% more nodes were visited in higher dimensions (> 4) in our experiments. In high dimensions, the number of visited nodes by MPES is fewer than that of MPTS R-trees, but the query processing time of MPES is slower than MPTS R-trees. This can be explained by R-tree node utilization. Although the number of visited nodes in MPES is fewer, the number of MBB comparisons using MPES is greater than MPTS R-trees because only 66% of maximum node fan-outs are used by R-tree nodes on average. In another word, although MPTS R-trees access global memory more often, the processing time of a memory block in MPTS R-trees is shorter than MPES because 33% of the memory block is empty in MPTS R-trees and the number of instructions executed by a warp is fewer.

Figure 3.8 shows average query response time for individual queries. The data-parallel partitioned indexing schemes shown in Figure 3.8b processes individual queries more than 100

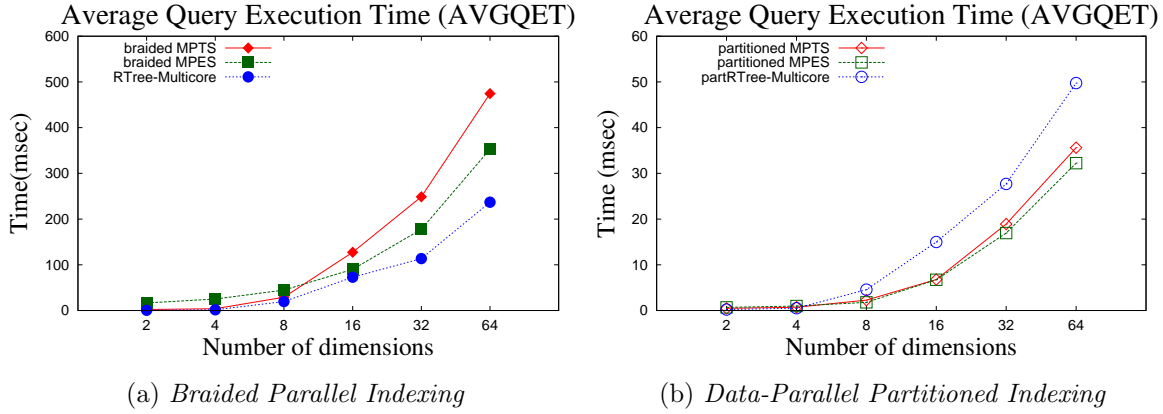


figure 3.8: Query Execution Time in High Dimensions

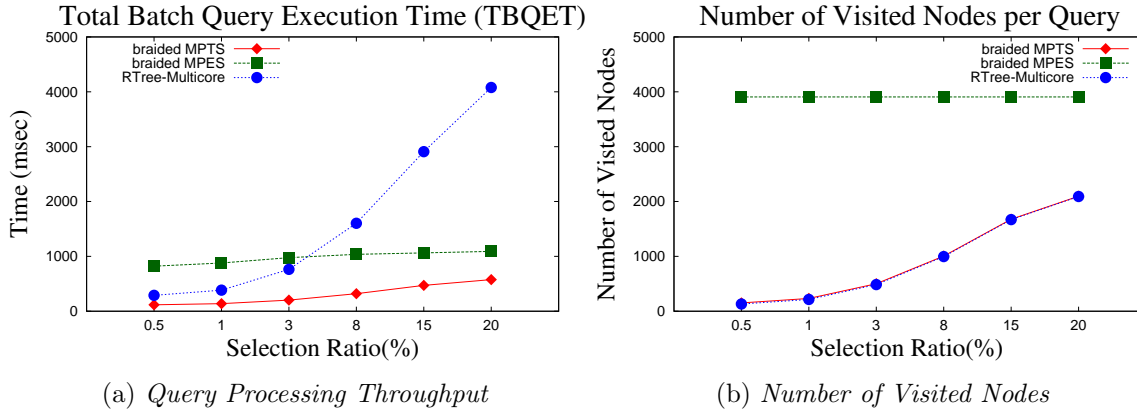


figure 3.9: Throughput and Number of Visited Nodes with Varying Selection Ratio

times faster than braided parallel indexing shown in Figure 3.8a, In Figure 3.8a, RTree-Multicore shows faster performance than braided parallel MPTS R-Trees and MPES in terms of single query processing, but MPTS R-Trees and MPES with data-parallel partitioned indexing on the GPU outperform both braided-parallel and data-parallel partitioned R-Trees indexing on multi-core CPUs (RTree-Multicore and partRTree-Multicore).

Figure 3.9 and 3.10 show how much MPTS search algorithm is sensitive to the *selection ratio* of queries. When the selection ratio is small, the distance between the leftmost overlapping leaf node and the rightmost overlapping leaf node will be likely to be short. In such a case the MPTS search will visit much fewer tree nodes than MPES.

In the experiments shown in Figure 3.9, we increase the selection ratio from 0.5% to 20%. The number of indexed data is 1 million and we submit 1,000 queries. Interestingly, the indexing scheme that suffers from increased selection ratio is not MPTS R-trees but RTree-Multicore on quad core Intel Xeon E5506. Again MPTS search algorithm visits almost the same number of tree nodes with recursive search algorithm, but the TBQET of MPTS R-Trees is much smaller than RTree-Multicore because MPTS R-Trees process them with a much larger number

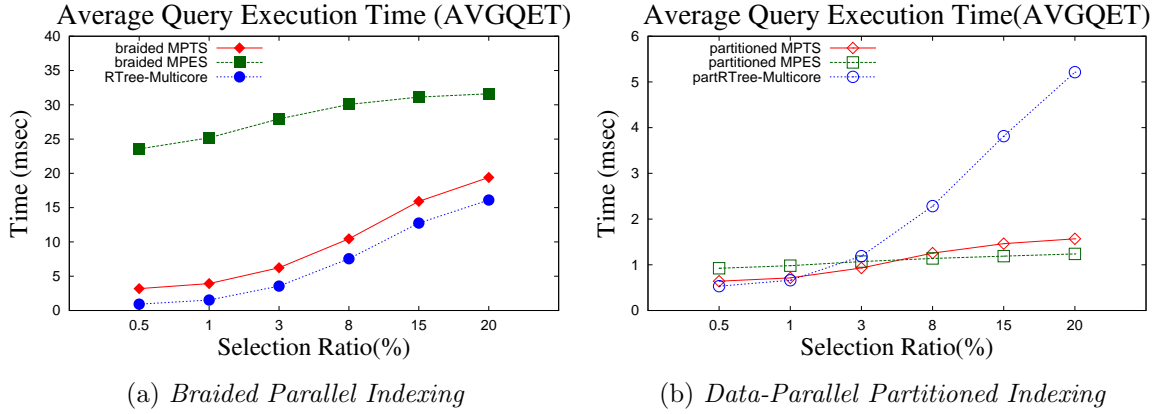


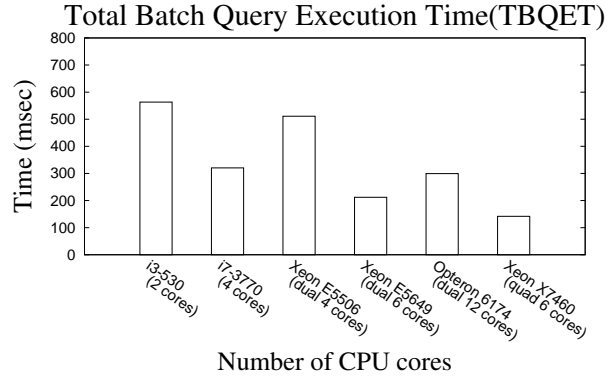
figure 3.10: Query Execution Time with Varying Selection Ratio

of cores. Throughout in our experiments, MPTS search algorithm has been consistently shown to effectively prune out non-overlapping leaf data. No matter how large or small the selection ratio is, the three-phase scanning algorithm of MPTS R-trees visits almost the same number of nodes with the regular R-tree search algorithm. In the worst case, about 15% more nodes were visited when the selection ratio was 0.5%, 15% more nodes when selection ratio was 1%, and less than 1% more nodes when selection ratio was higher than 1%.

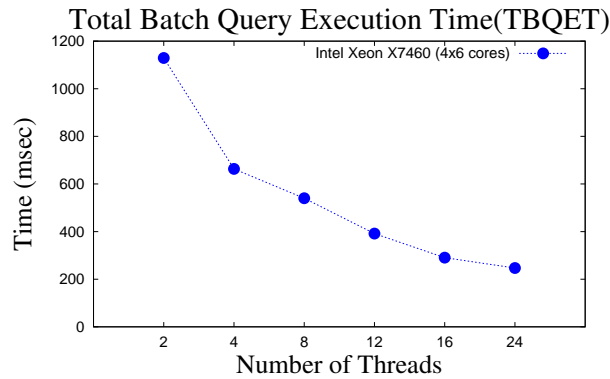
When the selection ratio is small, brute-force MPES does not perform very well. But as the selection ratio increases, the number of tree nodes to visit increases and RTree-Multicore becomes slower than MPES. When the selection ratio is 20%, the query processing throughput of MPES is about 4 times higher than multi-threaded R-trees (RTree-Multicore) and MPTS R-trees yields about 7 times higher system throughput than RTree-Multicore.

Figure 3.10 shows the data-parallel partitioned indexing helps accelerate individual query processing. In data-parallel partitioned indexing, MPES processes queries even slightly faster than MPTS R-trees when the selection ratio is higher than 8%, this is because of the overhead of leftmost and rightmost search phases. It should be noted that the query processing time of MPES slightly increases as selection ratio increases. This is not because MPES scans more number of data with higher selection ratio, but it is because of the function that compares the MBB overlap of a node and a query. The function stops comparing coordinates of MBBs immediately when they do not overlap. But if they overlap, all the coordinates have to be checked, so it takes more time with higher selection ratio.

In the set of experiments shown in Figure 3.11a, we evaluate the performance of multi-threaded R-trees (RTree-Multicore) on various different multi-core CPUs. The number of threads is determined by the number of cores, i.e., we run 4 threads for a single socket quad core CPU (i7-3770), 8 threads for dual socket quad core CPU (Xeon E5506), 12 threads for dual socket 6 core CPUs (Xeon E5649), 24 threads for dual socket 12 core CPU (Opteron 6174), and



(a) Throughput on various CPUs



(b) Throughput with varying number of threads on Intel Xeon X7460 (24 cores)

figure 3.11: Performance evaluation on various multi-core architectures

quad socket 6 core CPU (Xeon X7460). The query processing throughput of quad Xeon X7460 (24 threads) is only 3 times greater than that of i3-530 (2 threads), and only 1.7 times faster than i7-3770 (4 threads). This result is showing that in multi-core architectures the number of cores is not determining the tree index search performance, but the processing power and clock speed of each core is also important. In the experiments shown in Figure 3.11b, we measure the query processing throughput with increasing number of threads on 24 core machines. The throughput increases sub-linearly as we run more concurrent threads, but it saturates when the number of threads is larger than the number of available cores.

In addition to the synthetic point datasets, we evaluate the proposed parallel indexing schemes using real spatial datasets - that contains one million two-dimensional MBBs of California streets. As shown in Figure 3.12, the result is not quite different from the synthetic datasets. MPTS R-trees consistently outperforms MPES and multi-threaded R-trees (RTree-Multicore). When the selection ratio is less than 3%, RTree-Multicore outperforms MPES, but when it is higher than 3%, MPES performs better. In Figure 3.12b, the number of visited node with MPES is about 3 times greater than that of RTree-Multicore, but MPES takes the advan-

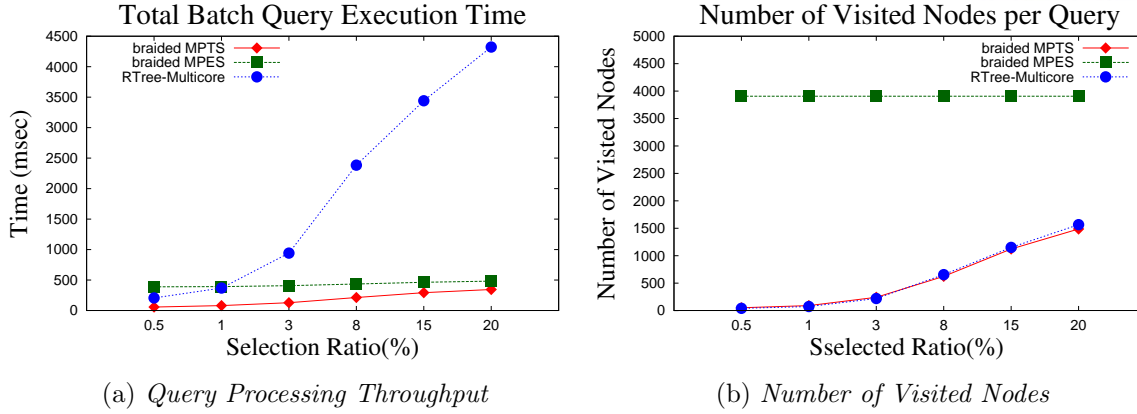


figure 3.12: Performance evaluation using San Francisco roadmap data

tage of massive parallelism to win. However still MPTS R-tree yields the highest throughput because it visits almost the same number of tree nodes with regular R-trees, and those nodes are visited in parallel by a large number of GPU processing units.

3.6 Summary

In this work, we proposed parallel multi-dimensional range query algorithm for GPU. The proposed scheme improves the utilization of GPU architecture for range query processing and avoids the irregular search path by transforming the tree traversal problem into a sequential data processing problem. Our experimental results demonstrate how the proposed algorithm - MPTS R-Trees effectively prunes out irrelevant tree nodes while it places very little overhead on the GPU. The search time of MPTS algorithm on the GPU Fermi M2090 is as low as 20% of parallel R-trees on quad-core Intel Xeon E5506 architecture and consistently outperforms brute-force scanning methods.

We have also compared the braided parallel indexing and data-parallel partitioned indexing, and presented experimental results that show braided parallel indexing improves system throughput when a large number of concurrent queries are submitted and data-parallel partitioned indexing helps improve individual query response time. We postulate the two parallel indexing schemes can be adaptively employed in the case when the query arrival distribution changes dynamically.

Chapter 4. Exploiting Massive Parallelism of the GPU for Multi-dimensional Indexing

In this chapter, we introduce a novel parallel tree traversal algorithm - *Massively Parallel Restart Scanning (MPRS)* for multi-dimensional range queries that avoids recursion and irregular memory access. The MPRS algorithm traverses hierarchical tree structures with mostly contiguous memory access patterns without recursion, which offers more chances to optimize the parallel SIMD algorithm. Our experiments show braided parallel MPRS range query algorithm achieves at least 80% SIMD efficiency while task parallel tree traversal algorithm shows only 9%-15% SIMD 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.

4.1 Existing Stackless Ray Traversal Algorithms

In computer graphics, stackless ray traversal algorithms have been proposed for efficient ray tracing on the GPU because a large number of rays are traced in parallel and the overhead of using run-time stack can be very high. Some of the stackless ray tracing algorithms can not be used for multi-dimensional range query processing directly, but there are other stackless search algorithms that can be adopted for multi-dimensional range queries with some modifications to the algorithms. For 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 chapter, we list a couple of stackless ray tracing algorithms and discuss the extensions of them we adopted for multi-dimensional range query processing.

4.1.1 Exhaustive Scanning on GPU

As we described earlier, the tree node traversal pattern is inherently irregular and recursive function call on the GPU is very slow. Therefore, instead of traversing spatial indexing structures recursively, we implemented a simple but massively parallel exhaustive search function as a CUDA kernel function to transform range query search into a streaming data filtering process, and to exploit SIMD execution model of CUDA architecture.

When the total number of multi-dimensional datasets to search is N , we simply divide it by a given number of GPU threads and each thread compares whether every single element of

the assigned dataset overlaps the given query.

4.1.2 User-defined Stack in Global Memory

Another 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 [60] 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.

4.1.3 Kd-restart for Range Query Processing

Kd-restart algorithm [27] eliminates the stack operations by restarting the search at the root of the tree. Instead of back-tracking, 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.

4.1.4 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 [63, 29]. Havran et al. [63] 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. [29] 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 MBB, more than one ropes should be traversed and thereby stack operations can not be eliminated.

4.1.5 Parent Link

Another simple strategy is to traverse the hierarchical tree structure as in graph traversal. Hapala et al. [28] 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.

4.1.6 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 4.1 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 search path visit always 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 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 leaf node D and G in Figure 4.1 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 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.

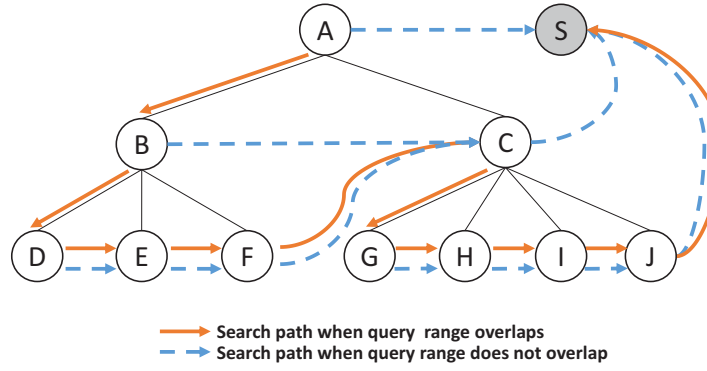


figure 4.1: *Query Processing with Skip Pointer*

4.1.7 Short Stack for R-tree

Horn et al. [31] 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 reducing the number of global memory accesses compared to 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.

4.2 Multi-dimensional Indexing Structures on the GPU

In scientific data analysis applications, the size of data sets is usually enormous but the number of submitted concurrent queries is relatively smaller than that of 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 cooperate in order to process a single query in parallel.

Multi-dimensional indexing structures 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, their 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.

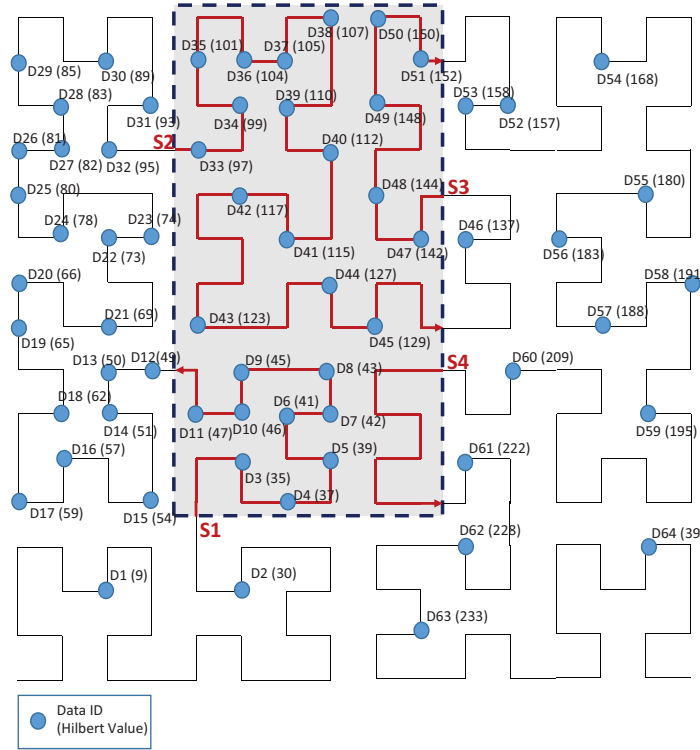


figure 4.2: *Hilbert Curve: Multi-dimensional range query overlaps some number of runs ($S1 \sim S4$) on Hilbert curve, i.e., the data objects that overlap a multi-dimensional query range are discontinuously stored.*

4.2.1 Massively Parallel Hilbert R-Tree (MPHR-Tree)

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 4.2 and Figure 4.3. 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 [61]. 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* [21]. 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 [64]. In order to sort the Hilbert values of the multi-dimensional data, we employed *Thrust* [65], 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 MBB of leaf nodes via parallel reduction and stores the MBB 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.

4.2.2 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 Massively Parallel Tree Traversal Algorithms

4.3.1 MPRS: Massively Parallel Restart Scanning

Massively Parallel Restart Scanning (MPRS) algorithm - a multi-dimensional range query processing algorithm we propose traverses the hierarchical tree structures from root node to leaf nodes multiple times as in kd-restart algorithm [27]. For a given range query, no matter how many MBB 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

Algorithm 6 *MPRS Range Query Algorithm*

void MPRS_RangeQueryKernel(Node* root, MBB query)

```

1: shared int Ovlp[NumberOfChildNodes];
2: int visitedLeafIdx  $\leftarrow$  0
3: for all tid  $\leftarrow$  1, numThreads do
4:   node  $\leftarrow$  root
5:   while visitedLeafIdx < root.maxLeafIdx do
6:     while node is an internal node do
7:       Ovlp[tid]  $\leftarrow$  INT_MAX
8:       // each thread compares a query with the bounding box of its child node
9:       if tid < node.numChildren and visitedLeafIdx < node.leafIdx[tid] and
       Overlap(query, node) then
10:        Ovlp[tid]  $\leftarrow$  tid // If an overlapping child node is unvisited
11:      end if
12:      syncthreads()
13:      // parallel reduction to find out the leftmost overlapping child
14:      leftmost  $\leftarrow$  parallelReduction(Ovlp)
15:      if leftmost == INT_MAX then
16:        // If there's no overlapping child node, update the visitedLeafIdx
17:        visitedLeafIdx  $\leftarrow$  node.maxLeafIdx // restart the search from root
18:        node  $\leftarrow$  root;
19:      else
20:        node  $\leftarrow$  node.child[leftmost] // fetch the leftmost child node
21:      end if
22:      syncthreads()
23:    end while
24:    while node is a leaf node do
25:      if tid < node.numChild and Overlap(query, node) then
26:        hitFlag  $\leftarrow$  true
27:        SaveOverlappingData(node.data[tid])
28:      end if
29:      visitedLeafIdx  $\leftarrow$  node.maxLeafIdx
30:      if hitFlag == true then
31:        node  $\leftarrow$  node.rightSibling // fetch next sibling node
32:      else
33:        node  $\leftarrow$  node.parent // parent check
34:      end if
35:    end while
36:  end while
37: end for

```

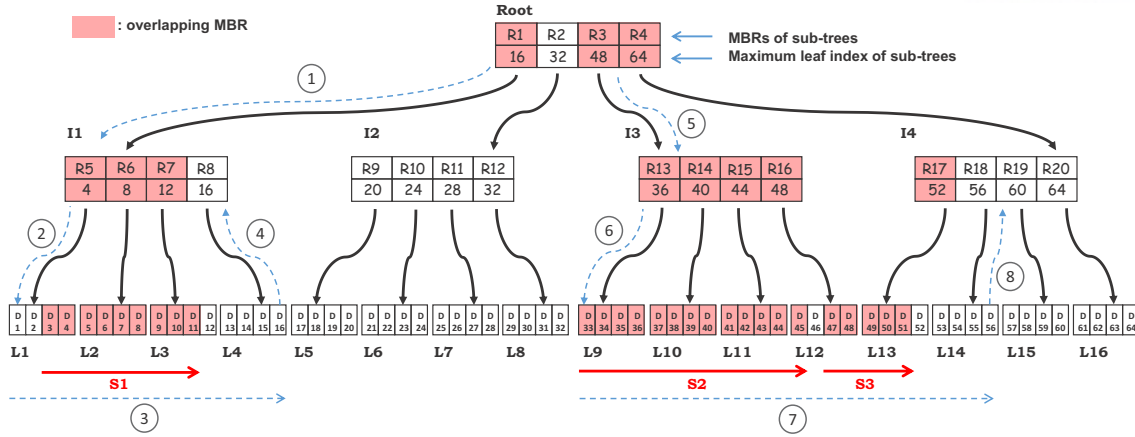


figure 4.3: *Massively Parallel Restart Scanning with MPHR-tree Structure*

because our MPRS algorithm does not backtrack to already visited tree nodes.

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 4.2 overlaps four Hilbert curve segments - S_1 , S_2 , S_3 , and S_4 .

The MPRS search algorithm finds out the smallest (in terms of the Hilbert curve index) multi-dimensional data object that overlaps a query range (D_3 in Figure 4.2). 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 ($D_4 \sim D_{11}$). While scanning data objects along the continuous segment on the Hilbert curve, it needs to jump to the starting position (D_{33}) of the next segment (S_2) on the Hilbert curve if it visits a non-overlapping data object (D_{12}).

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 tree traversal, MPRS search algorithm uses the leaf index stored in tree nodes in order to avoid visiting already visited leaf nodes. In 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 ~ 27%.

The detailed MPRS algorithm is described in Algorithm 6. *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 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.

Figure 4.3 shows an example of MPHR-tree and the MPRS search path for the data objects and the query illustrated in Figure 4.2. In the beginning, *visitedLeafIdx* is set to 0 and each

thread compares the MBB of each child node with a query range. Suppose the query range overlaps MBB - $R1$, $R3$, 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 MBB 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 MPRH-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 MBB 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 MBB $R4$ 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 MBB 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 [66], 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.

4.4 Evaluation

4.4.1 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 4.2.1 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 [67]. 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 4.1 and 4.2.1.

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. Note that the GPU kernel launching overhead and data transfer time are not included in the average query response time. 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.

4.4.2 Experimental results

Parallel Construction of MPHR-trees

table 4.1: 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.

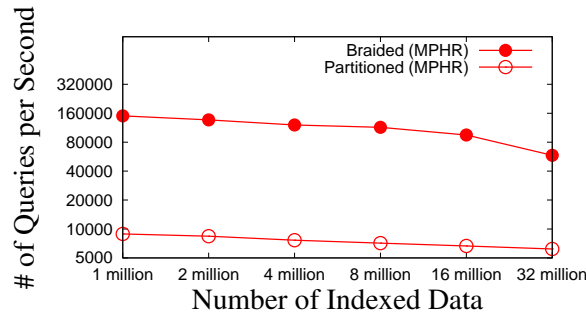
Table 4.1 shows the elapsed time of constructing MPHR-trees in a bottom-up fashion on a single NVIDIA M2090 GPU. In the experiments, the MPHR-tree construction spends more than 77% of its total construction time (2.866 seconds for 40 millions of data) on sorting Hilbert values, while the bottom-up parallel 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

¹On AMD Opteron 6128 CPU, it takes 1.72 seconds to construct the MPHR-trees with 40 million data objects

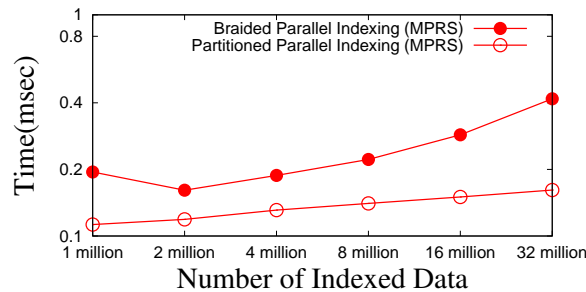
multi-dimensional index for 40 millions of data on an NVIDIA M2090 GPU.

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 [32]. 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

figure 4.4: *Query Processing Performance With Braided Parallel Index and Partitioned Index*

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.

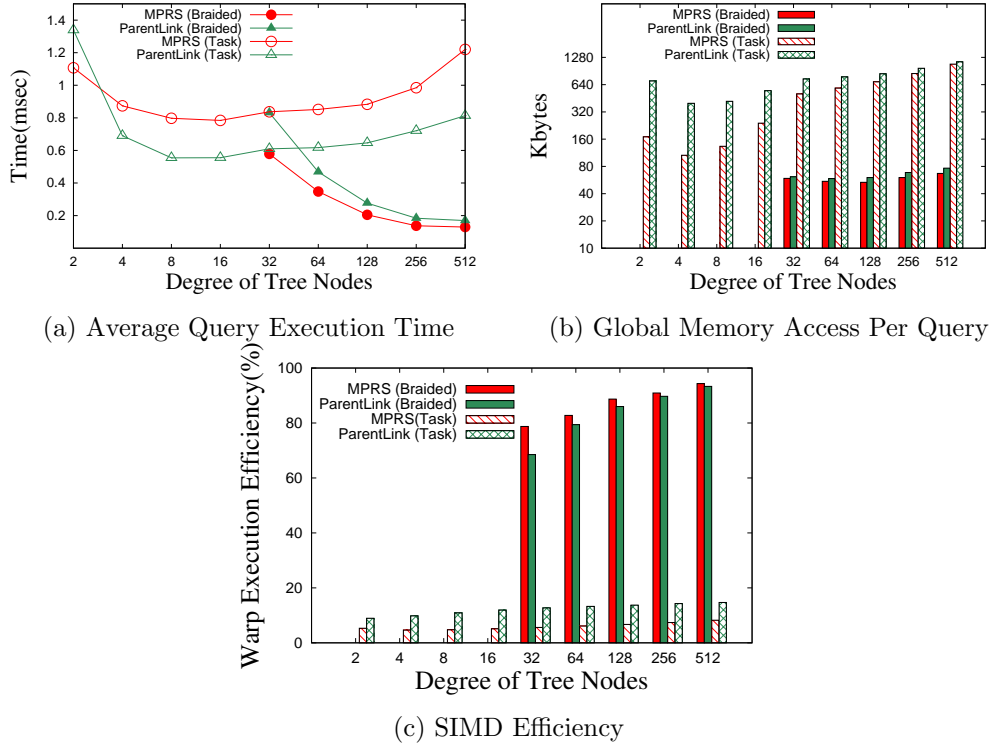


figure 4.5: *Search Performance with Varying the Degree of Tree Nodes*

In the experiments shown in Figure 4.4, 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 queries can be processed in parallel on a single GPU using braided parallel indexing.

Varying Degree of Tree Nodes

In the experiments shown in Figure 4.5, 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 4.1.6, 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 MBB 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 4.5b. 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 4.5b 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 4.5c 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 [44]. 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.

MPHR-Trees vs. disjoint MSP BVH

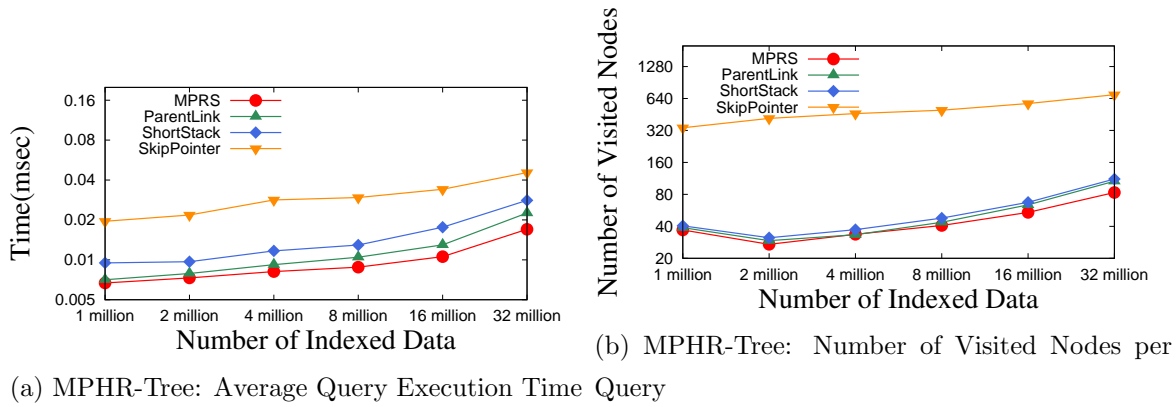


figure 4.6: *Average Query Response Time with MPHR-Tree*

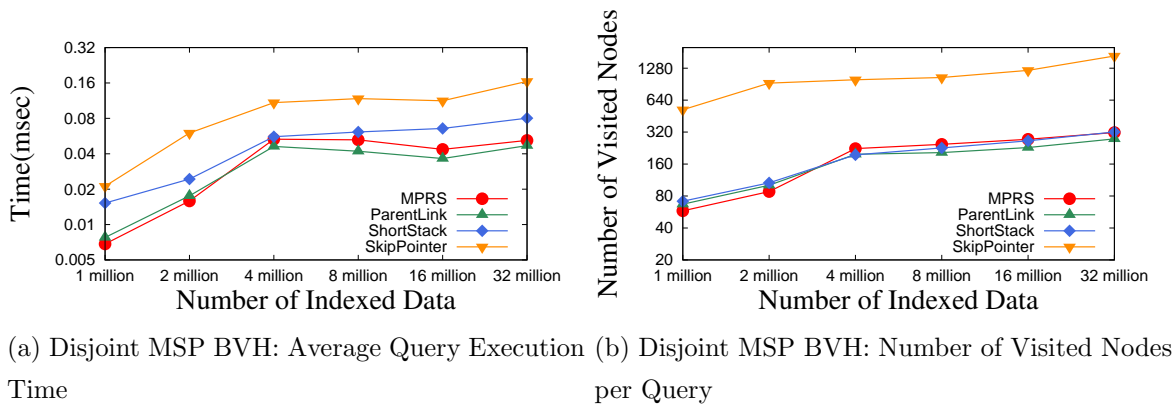


figure 4.7: *Average Query Response Time with disjoint MSP BVH*

In Figure 4.6 and Figure 4.7, we compare the stackless multi-dimensional range query processing algorithms with two braided parallel indexing schemes - MPHR-tree shown in Figure 4.6a and disjoint MSP BVH shown in Figure 4.7a. 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 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.

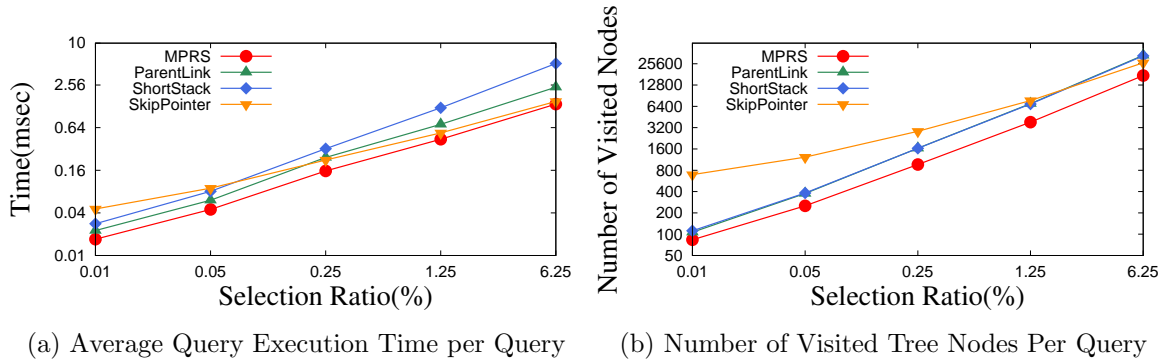


figure 4.8: *R-Tree: Search Performance with Varying Selection Ratio*

Selection Ratio

In the experiments shown in Figure 4.8, 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 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.

Performance in High Dimensions

For the last set of experiments shown in Figure 4.9, 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.

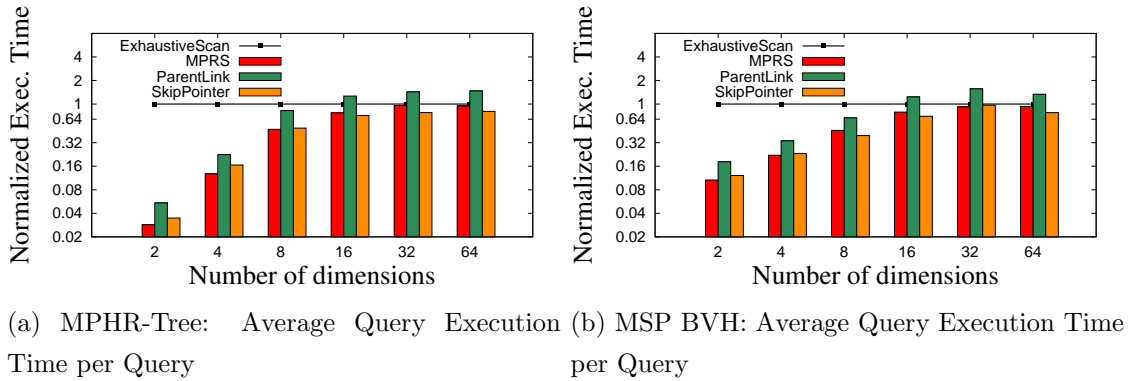


figure 4.9: *Search Performance with Varying Number of Dimensions*

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 [58]. 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

²For 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.

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 4.9, 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 4.9b, 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 4.9a 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.

4.5 Summary

In this work, we present 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.

Chapter 5. Co-Processing Heterogeneous Parallel Index

Although we successfully leverage a large number of cores in the GPU for multi-dimensional indexing, we found that the GPU-based tree-structured indexing has some limitations. First, the internal node traversal on the GPU still suffers from the *warp-divergence* problem. Another important limitation 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 GPUs to directly access host memory via NVLink [68]. 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.

To address these limitations, 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, this work proposes to scan a large number of leaf nodes in parallel according to the selection ratio of the range query. For such parallel scanning, the GPU is known to be superior to the CPU.

The experimental results show that our proposed multi-dimensional range query co-processing scheme improves the query response time by up to 12x and query processing throughput by up to 4x compared to the state-of-the-art GPU tree traversal algorithm.

5.1 Hybrid Tree Structure

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 5.1. The upper tree is a traditional multi-dimensional tree-structured index that resides in the CPU host memory, 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

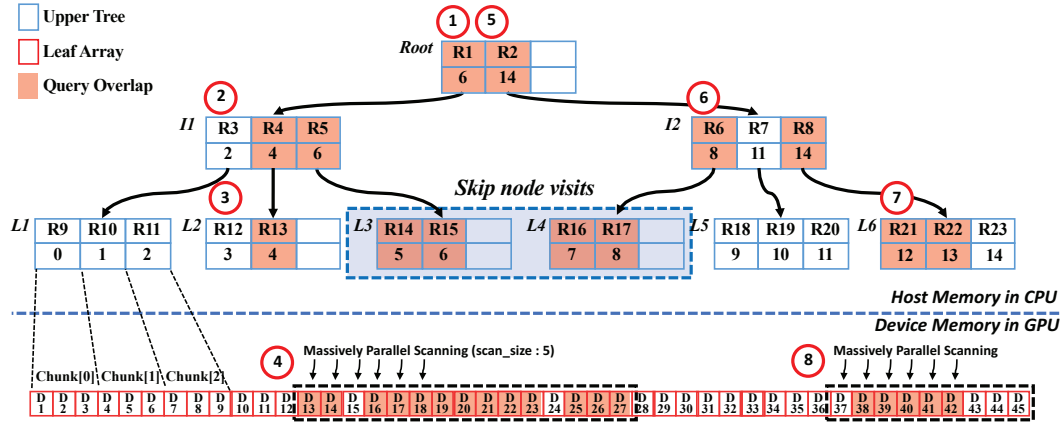


figure 5.1: An example of a co-processing Hybrid tree with CPU and GPU: Leaf node scanning on the GPU (step 4) overlaps the next tree traversal on the CPU in time(steps 5-7)

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* [61] 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 [69, 11].

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

Algorithm 7 *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

```

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.

5.2 Query Co-Processing in 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, 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 7, 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 5.1, 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 7. 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 5.1 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 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

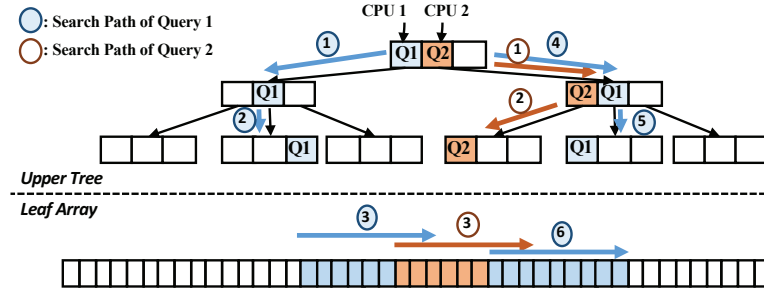


figure 5.2: Multiple Query Scheduling

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.

5.3 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 5.2. When a thread reaches the parent node of a leaf node (step ② in Figure 5.2), 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 [70], 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.

5.4 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) [71]. 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 [24], 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.

5.4.1 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%).

5.4.2 GPU Kernel Launch Overhead

table 5.1: 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

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 5.1, 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

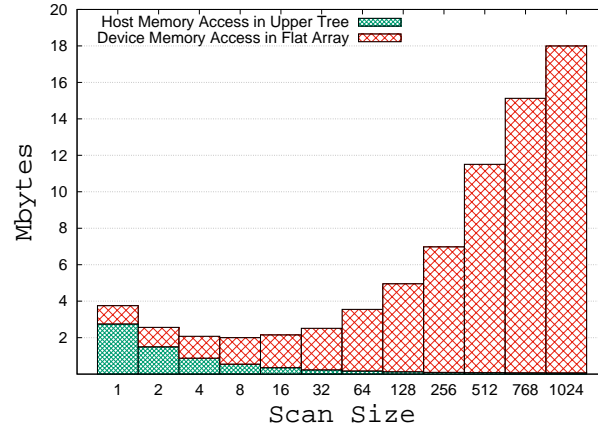


figure 5.3: Amounts of Host and Device Memory Access with Varying Scan Size

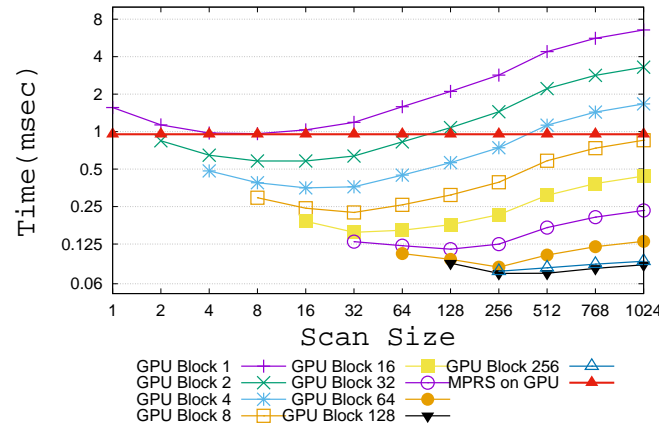


figure 5.4: Query Execution Time of Various GPU Thread Blocks and Scan Size

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 (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.

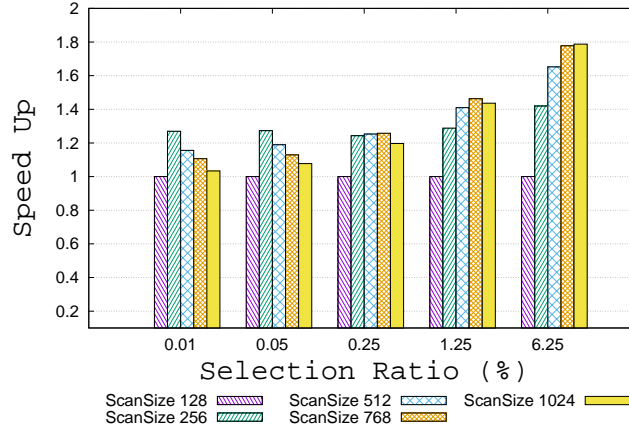


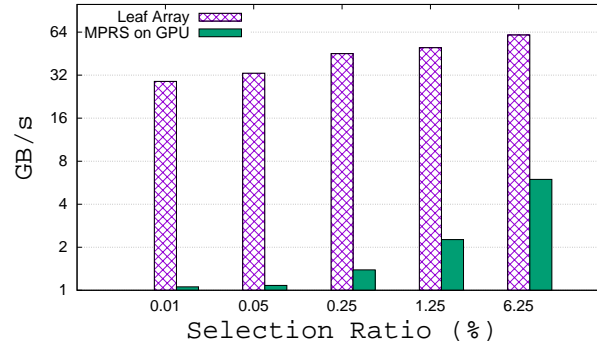
figure 5.5: Speed Up with Various Selection Ratio and Scan Size(Normalized to 128)

5.4.3 Various GPU Thread Block & Scan Size

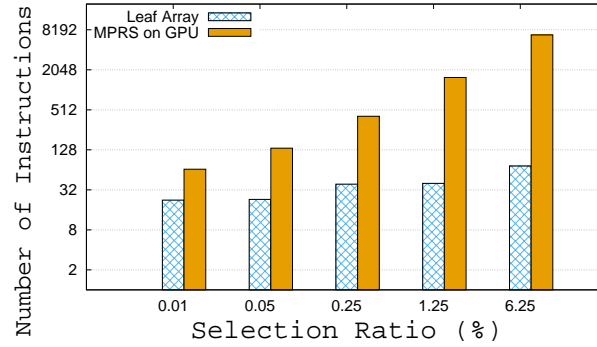
In the experiments shown in Figure 5.3 and Figure 5.4, we index 100 million three-dimensional data points and measure the amount of memory accessed in accordance with varying `scan_size` and the number of GPU blocks and the average query response time of 1,000 queries. When the scan size is set to a 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 kernel function for each leaf node. As shown in Figure 5.3, 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 points, the GPU visits too many non overlapping data points, which also affects the query response time. As shown in Figure 5.3, 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



(a) Global Memory Load Throughput per Second



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

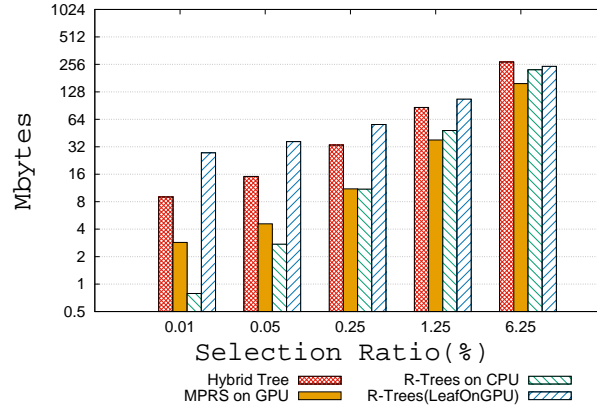
figure 5.6: Profiled Results with Various Selection Ratio

query co-processing.

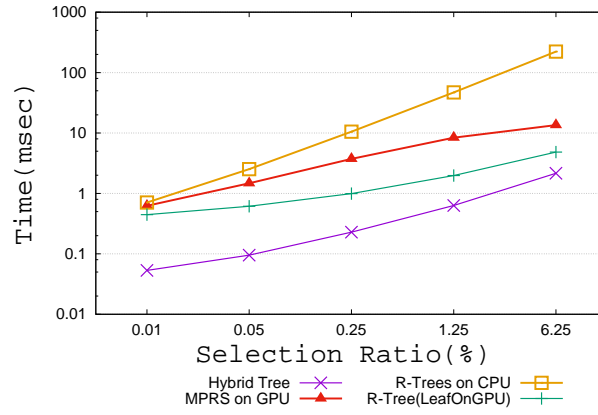
When we increase the number of GPU blocks instead of increasing the number of data points per thread, it better utilizes a large number of streaming multiprocessors in the GPU and the query processing performance improves. Note that multiple thread blocks concurrently access different parts of the leaf 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 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 5.4 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,



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



(b) Average Query Execution Time with Various Selection Ratio per Query

figure 5.7: Performance Results with Various Selection Ratio

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 [24]. 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 5.5, 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

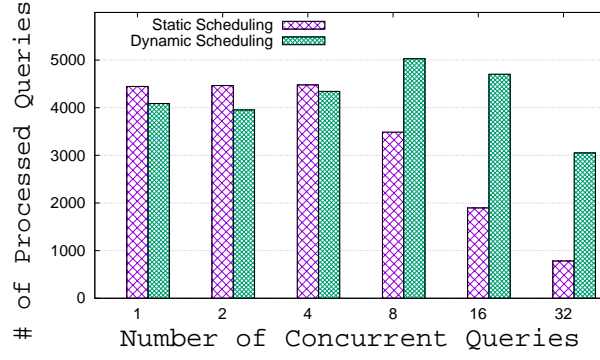
result in better performance. When the selection ratio is lower than 0.25%, a large `scan_size` performs poorly because it increases the number of unnecessarily accessed data points that do not overlap.

Figure 5.6a 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 ~ 61 GB/s, that of the MPRH-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.

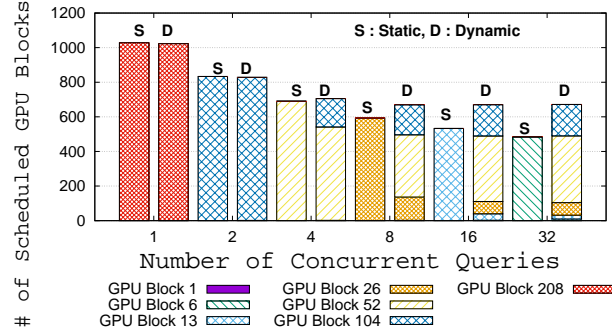
Figure 5.6b shows the number of average conditional branch instructions executed per query. As we increase the selection ratio, more tree nodes are accessed and more `if` statements are called. When the selection ratio is 6.25%, the MPRS algorithm requires a 94x number of conditional branches, which degrades the global memory load throughput and query response time.

Figure 5.7a 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 5.6a. 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 MPRH-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 5.7b 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)



(a) Query Throughput per Second



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

figure 5.8: Static vs. Dynamic Query Scheduling

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.

5.4.4 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 5.8 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 5.3. 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 5.8b 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.

5.5 Summary

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.

Chapter 6. Conclusion

In this dissertation, we investigate the problem of 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 the recursion and irregular tree access patterns. To address this problem, we propose a tree traversal algorithm - *MPTS* that does not require the stack operation and access tree nodes in a sequential manner. The experimental results show that our MPTS tree traversal algorithm consistently outperforms traditional recursive R-Tree search algorithm for multi-dimensional range query processing.

To reduce query response time, we propose a novel parallel multi-dimensional indexing structure, *MPHR-trees* and tree traversal algorithm *MPRS*. The 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. We showed that braided parallel indexing outperforms task parallel indexing in terms of the amount of global memory accesses and SIMD efficiency and MPHR-tree outperforms disjoint multi-way MSP bounding volume hierarchy because of high node utilization in low level tree nodes.

Furthermore, we present 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. The experimental results show that our proposed multi-dimensional range query co-processing scheme improves the query response time by up to 12x and query processing throughput by up to 4x compared to the state-of-the-art GPU tree traversal algorithm.

The multi-dimensional range query processing algorithm is one of the most important problems in various fields including high performance computing. For example, in computer graphics, GPUs accelerate 3D image rendering, and some image rendering techniques that can utilize 3D index such as ray tracing, collision detection, ambient occlusion, and volume visualization, will take the advantages of an efficient multi-dimensional range query processing on the GPU.

Bibliography

- [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, and S. M. LaConte, “Medical image processing on the GPU – past, present and future,” *Medical Image Analysis*, vol. 17, no. 8, pp. 1073 – 1094, 2013.
- [3] 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*, vol. 28, no. 16, pp. 2618–2640, Dec. 2007.
- [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 kelmholtz 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, and D. Kaeli, “Exploiting memory access patterns to improve memory performance in data-parallel architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2011.
- [6] J. Kim and B. Nam, “Parallel multi-dimensional range query processing with r-trees on GPU,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1195–1207, 2013.
- [7] 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*, vol. 26, no. 1, pp. 80–113, 2007.
- [8] W.-K. Jeong and R. T. Whitaker, “A fast iterative method for eikonal equations,” *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2512–2534, July 2008.
- [9] J. Kruger and R. Westermann, “Acceleration techniques for gpu-based volume rendering,” in *Proceedings of the 14th IEEE Visualization 2003 (VIS’03)*. IEEE Computer Society, 2003, p. 38.
- [10] P. Harish and P. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *HiPC*, vol. 7. Springer, 2007, pp. 197–208.
- [11] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for gpu computing,” in *Graphics hardware*, vol. 2007, 2007, pp. 97–106.

- [12] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. IEEE, 2004, pp. 47–47.
- [13] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 195–206.
- [14] R. Rew, G. Davis, and S. Emmerson, "NetCDF User's Guide for C," 1997, <http://www.unidata.ucar.edu/packages/netcdf/cguide.pdf>.
- [15] M. Folk, "A White Paper: HDF as an Archive Format: Issues and Recommendations," January 1998, <http://hdf.ncsa.uiuc.edu/archive/hdfasarchivefmt.htm>.
- [16] 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.
- [17] 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.
- [18] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, 3rd ed. Natick, MA, USA: A. K. Peters, Ltd., 2009.
- [19] C. Ericson, *Real-Time Collision Detection*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.
- [20] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, vol. 19, no. 2. ACM, 1990, pp. 322–331.
- [21] 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)*, 1994, pp. 500–509.
- [22] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects." Tech. Rep., 1987.
- [23] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An index structure for high-dimensional data," in *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, 1996, pp. 28–39.

- [24] J. Kim, W.-K. Jeong, and B. Nam, “Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu,” *Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, 2015.
- [25] V. Garcia, E. Debreuve, and 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.
- [26] V. Garcia, E. Debreuve, F. Nielsen, and 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.
- [27] T. Foley and J. Sugerman, “KD-tree acceleration structures for a gpu raytracer,” in *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005.
- [28] M. Hapala, T. Davidov, 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.
- [29] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek, “Stackless KD-tree traversal for high performance GPU ray tracing,” in *Eurographics*, 2007.
- [30] B. Smits, “Efficiency issues for ray tracing,” *Journal of Graphics Tools*, vol. 3, no. 2, pp. 1–14, 1998.
- [31] 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.
- [32] 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.
- [33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron., “A performance study of general purpose applications on graphics processors using cuda.” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.
- [34] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, “Relational query co-processing on graphics processors,” *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 21–39, 2009.
- [35] P. Bakkum and K. Skadron, “Accelerating SQL database operations on a GPU with CUDA.” in *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

- [36] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha, “Fast computation of database operations using graphics processors,” in *Proceedings of 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2004.
- [37] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, “Bio-sequence database scanning on a gpu,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 8–pp.
- [38] B. Nam and A. Sussman, “Spatial indexing of distributed multidimensional datasets,” in *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2005.
- [39] —, “Analyzing design choices for distributed multidimensional indexing,” *Journal of Supercomputing*, vol. 59, no. 3, pp. 1552–1576, 2012.
- [40] I. Kamel and C. Faloutsos, “Parallel R-trees,” in *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1992, pp. 195–204.
- [41] N. Koudas, C. Faloutsos, and I. Kamel, “Declustering spatial databases on a multi-computer architecture,” in *Proceedings of the 5th International Conference on Extending Databases Technology (EDBT)*, 1996.
- [42] B. Schnitzer and S. T. Leutenegger, “Master-Client R-Trees: A new parallel R-tree architecture,” in *Proceedings of 11th International Conference on Scientific and Statistical Database Management (SSDBM)*, 1999, pp. 68–77.
- [43] B. Nam, M. Shin, H. Andrade, and A. Sussman, “Multiple query scheduling for distributed semantic caches,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 598–611, 2010.
- [44] Nvidia Corporation, “NVIDIA CUDA Compute Unified Device Architecture,” <http://www.nvidia.com/>.
- [45] 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.
- [46] 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.

- [47] 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.
- [48] L. Luo, M. D. Wong, and L. Leong, “Parallel implementation of r-trees on the GPU,” in *17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.
- [49] S. Laine, “Restart trail for stackless bvh traversal,” in *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010, pp. 107–111.
- [50] D. M. Hughes and I. S. Lim, “Kd-jump: a path-preserving stackless traversal for faster iso-surface raytracing on gpus,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, 2009.
- [51] A. Shahvarani and 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.
- [52] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, “idistance: An adaptive b+-tree based indexing method for nearest neighbor search,” *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 364–397, 2005.
- [53] N. Bhatia *et al.*, “Survey of nearest neighbor techniques,” *arXiv preprint arXiv:1007.0085*, 2010.
- [54] V. Garcia and F. Nielsen, “Searching high-dimensional neighbours: Cpu-based tailored data-structures versus GPU-based brute-force method,” in *4th International Conference on Computer Vision/Computer Graphics Collaboration Techniques, MIRAGE*, 2009.
- [55] L. Cayton, “Accelerating nearest neighbor search on manycore systems,” in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.
- [56] Y. Su, G. Agrawal, and J. Woodring, “Indexing and parallel query processing support for visualizing climate datasets,” in *Proceedings of 41st International Conference on Parallel Processing*, 2012, pp. 249–258.
- [57] R. Weber, H.-J. Schek, and S. Blott, “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces,” 1998.
- [58] R. E. Bellman, *Adaptive Control Processes: A Guided Tour*. NJ: Princeton University Press, 1961.

- [59] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, “An optimal algorithm for approximate nearest neighbor searching fixed dimensions,” *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, Nov. 1998.
- [60] 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.
- [61] 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*, vol. 13, no. 1, pp. 124–141, 2001.
- [62] Y. Theodoridis, “R-tree Portal,” <http://www.rtreeportal.org>.
- [63] V. Havran, J. Bittner, and J. Zara, “Ray tracing with rope trees,” in *14th Spring Conference on Computer Graphics (SCCG)*, 1998.
- [64] 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.
- [65] NVIDIA, “Thrust,” <https://developer.nvidia.com/thrust>.
- [66] H. Jagadish, “Linear clustering of objects with multiple attributes,” in *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1990.
- [67] 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.
- [68] D. Foley, “Nvlink, pascal and stacked memory: Feeding the appetite for big data,” *Nvidia.com*, 2014.
- [69] N. Govindaraju, J. Gray, R. Kumar, and 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.
- [70] B. Nam, M. Shin, H. Andrade, and A. Sussman, “Multiple query scheduling for distributed semantic caches,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 598–611, 2010.

- [71] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, “Fast bvh construction on gpus,” in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.