Parallel Multi-dimensional Range Query Processing with R-Trees on GPU

Jinwoong Kim [a] Beomseok Nam [a]

[a] *School of Electrical and Computer Engineering*
Ulsan National Institute of Science and Technology
Ulsan, 689-798, Korea

Keyword

CUDA; GPGPU; GPU; Parallel indexing; Parallel multi-dimensional indexing; Parallel R-tree;

Abstract

The general purpose computing on graphics processing unit (GP-GPU) has emerged as a new cost effective parallel computing paradigm in high performance computing research that enables large amount of data to be processed in parallel. Large scale scientific data intensive applications have been playing an important role in modern high performance computing research. A common access pattern into such scientific data analysis applications is multi-dimensional range query, but not much research has been conducted on multi-dimensional range query on GP-GPU. Inherently multi-dimensional indexing trees such as R-Trees are not well suited for GPU environment because of its irregular tree traversal. Traversing irregular tree search path makes it hard to maximize the utilization of massively parallel architectures. In this paper, we propose a novel *MPTS (Massively Parallel Three-phase Scanning)* R-tree traversal algorithm for multi-dimensional range query, that converts recursive access to tree nodes into 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.

1 Introduction

Modern GPUs that have hundreds of processing units are being successfully and widely used as high performance accelerators for many general-purpose computations in various fields. Since NVIDIA produced a chip that is capable of programmable shading in early 2000s, the programmable shaders quickly became as flexible as CPUs while the shader pipeline of GPUs are yielding orders of magnitude higher performance than CPUs for vector processing. Now it is common to employ GPU as a modified form of stream processor. CUDA (Compute Unified Device Architecture) programming model allows programmers to run parallel algorithms on GPU's stream processors that can take the advantage of data-parallelism or task-parallelism while running serial portion of the algorithms simultaneously on CPU. Although NVIDIA keeps improving GPU architecture and CUDA programming model so that application programmers write general-purpose parallel programs on GPU, still GPU has many restrictions such as very small runtime stack size that make it difficult to convert various sequential algorithms into parallel algorithms with parallel random access memory. However, GPU draws lots of attention due to its significant speedups over single threaded program.

The massively parallel GPU architectures and their high-performance floating point arithmetic operations have been shown well-suited for large scale scientific data analysis applications. In many scientific disciplines, sensor devices and simulators generate truly large amounts of multi-dimensional datasets and the datasets are growing in size every day. Accessing and analyzing such large datasets is an important class of problems in data-intensive computing, and multi-dimensional indexing structures have played an important role in accelerating the access to subsets of such large datasets since multi-dimensional range query is one of the most common access patterns into scientific datasets. In order to handle multidimensional range queries efficiently, a large number of efficient and scalable indexing structures such as R-trees [12], R*-trees [3], and Hybrid-trees [7] have been proposed and improved.

However, not much research has been conducted to deploy multi-dimensional indexing structures on top of GPU architecture. Traversing spatial indexing structures on GPU is not an easy problem due to its irregular tree search paths of multi-dimensional range queries. Irregular memory access in CUDA may hurt efficient utilization of GPU's memory bandwidth, and recursive function calls to handle such irregular tree search paths can cause memory problems when a query has to visit a large number of tree nodes.

There have been some effort to employ SIMD (Single Instruction Multiple Data) techniques to efficiently search B+-Trees [27], but to the best of our knowledge not much effort has been made to parallelize multi-dimensional range query algorithm of R-Trees in a SIMD fashion. In [27], Zhou et al. described how SIMD (Single

Instruction Multiple Data) techniques can be employed to efficiently search B+-trees. The basic idea was to make SIMD instruction compare multiple keys at the same time. However, this approach does not work for processing multi-dimensional range queries in GPU architecture. With a given internal tree node and a multi-dimensional range query, more than one child node may overlap the given query and the sub-trees of all the overlapping child nodes have to be visited recursively. The number of overlapping child nodes to visit is not known priori, hence typically range query algorithms of multi-dimensional indexing structures use recursion or user-defined stack to visit a large number of nodes for the irregular search paths. However current GPU architectures have very small runtime stack size and accessing user-defined stack in global memory is intolerably slow.

In GPU architecture, algorithm performance is determined by how much portion of the algorithm can be parallelized. Without eliminating the irregular recursive tree traversal and knowing how many child tree nodes have to be visited priori, a large number of processing units in GPU can not be well utilized and it is difficult to efficiently navigate spatial indexing structures in parallel. Therefore, instead of the irregular recursive range query algorithms, we propose in this paper *Massively Parallel Three-phase Scanning* algorithm that effectively prunes out irrelevant tree nodes and efficiently serves multi-dimensional range queries on GPU.

Efficient multi-dimensional indexing structures on GPU will be beneficial not only for scientific applications but for computer graphics as well. 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 GPU.

The rest of the paper is organized as follows. In Section 2 we discuss other works related to the multi-dimensional indexing on GPU. In Section 3 we propose a new range query algorithm for GPU architecture, and discuss experimental results in Section 4. In Section 5 we conclude and discuss future work.

2 Related Work

Since GPU has been repeatedly reported that it offers unprecedented performance in various applications, GPU has also been studied to improve the performance of database SQL processing in addition to the index parallelization [2, 8, 13]. Bakkum et al. have implemented SQLite database virtual machine on GPU, and improved the performance of SELECT queries [2]. They transformed the internal B+-tree implementation of database table of SQLite into a straightforward row-column format in order to accelerate the query processing on GPU. Che et al. [8] reimplemented a set of computationally demanding general purpose applications on 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 [13]. Govindaraju et al. [11] also proposed GPU-aware algorithms for several common database operations such as conjunctive selections, aggregations, and semi-linear queries.

In spatio-temporal database community, there has been extensive research on multi-dimensional indexing tree structures, starting with the seminal work on R-trees [12]. 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 sub-tree, 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 [21, 23]. Kamel proposed Multiplexed R-trees [15], Koudas et al. proposed Master R-trees [18], 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. [24]. Nam et al. compared the challenges, and problems of designing distributed multi-dimensional indexes for data-intensive scientific applications [23]. The distributed multi-dimensional indexing structures have numerous usages in various contexts including distributed and parallel query processing systems [22].

As multi-core architectures have evolved, a couple of recent efforts were made to exploit SIMD execution of GPU to improve database query performance. Zhou et al. proposed to compare multiple keys of B+-trees at the same time using SIMD instruction [27]. Kaldewey et al. [14] 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 [16]. 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 different from our work in that they 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 algorithm.

Although GPUs have a large number of processing units, each processing units of high-performance GPUs is known to run a lot slower than a CPU core. Hence, some efforts have been made to improve the query processing throughput instead of reducing response time of each query. Fix et al. proposed *braided parallelism* for

traversing B+-Trees, wherein a single B+-Tree exists in GPU memory and multiple independent queries are concurrently executed within each SMP [9]. They showed this approach yields higher query processing throughput, but it does not help improve individual query response time. In this paper, we adapted the braided parallelism to improve the multi-dimensional query processing throughput, and compare it with pure data-parallel partitioned indexing scheme.

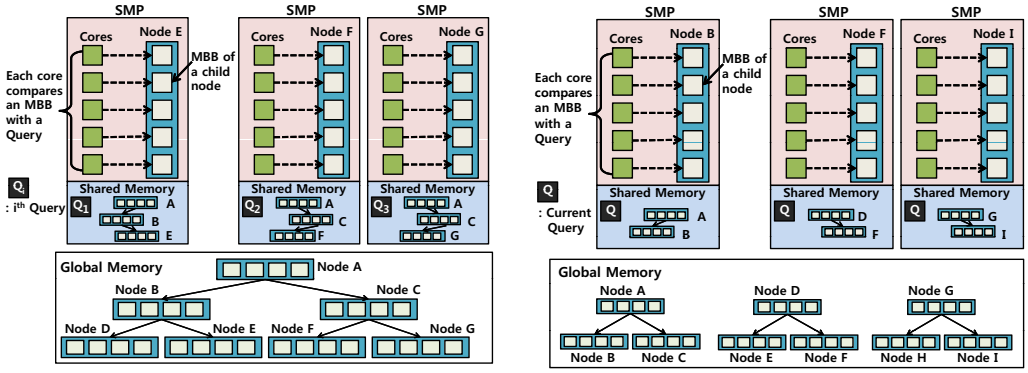
Nearest neighbor query (k-NN) is another important class of multi-dimensional queries that finds closest points given a set of points in multi-dimensional space. Garcia et al. [10] have proposed brute-force k-NN search algorithms for GPU, and Cayton presented a two-level brute-force search algorithm called *Random Ball Cover (RBC)* [6]. 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 relatively low dimensional range queries and proposes a novel tree traversing algorithm for hierarchical tree structures known to be very effective for low dimensional datasets.

Luo et al. [19] proposed a parallel R-tree traversal algorithm on GPU. Their work is similar to ours in a sense that they also tried to avoid irregular memory access and recursion by employing a queue in the shared memory of SMP. 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 proposed work is more scalable since it does not depend on the size of shared memory.

3 CUDA-enabled parallel spatial indexing structures

3.1 Background on CUDA

In order to process a large amount of data in parallel, a CUDA program spawns thousands of extremely lightweight parallel threads, and they execute the same *kernel function* on the GPU device 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 (48 KB in Tesla M2090) but low latency shared memory. In addition to the small shared memory, CUDA-enabled GPU cards come with a very large DRAM memory, called *global (device) memory*, which is shared and accessed by all CUDA threads.



(a) Braided Parallel Indexing (b) Data Parallel Partitioned Indexing

Fig. 1. *Braided Parallel Indexing vs Data Parallel Partitioned Indexing*

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 M2090 GPU contains 32 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 applications since it fits nicely with multi-SIMD architecture of GPU. However braided parallelism does not scale when the number of submitted tasks is small.

Since maximizing the utilization of GPU processing units plays key role in improving the performance of CUDA 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

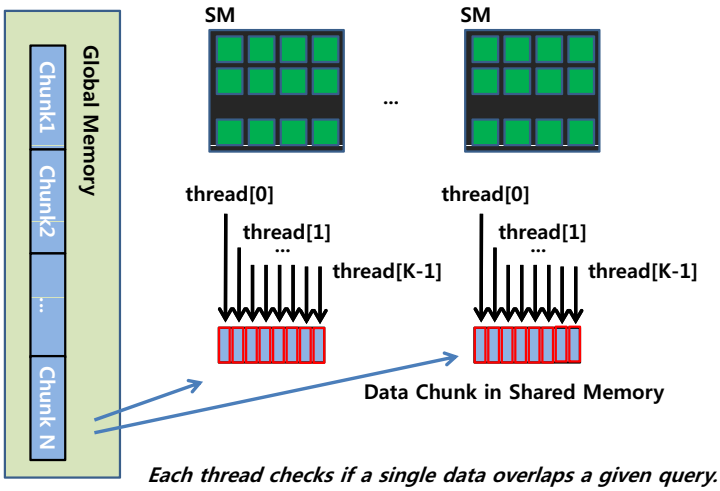


Fig. 2. *Massively Parallel Exhaustive Scanning on GPU (MPES)*

reduce the execution time of running each query.

In order to improve the response time of 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 1 illustrates the differences between braided parallel indexing scheme and data parallel partitioned indexing scheme. In order to refer to the data parallel partitioned indexing, we use the term partitioned indexing for short. In braided parallel indexing shown in Figure 1(a), each SMP processes different user query, hence if 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 1(b), 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.

3.3 Massively Parallel Exhaustive Scanning (MPES) on 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 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 3.4.

Algorithm 1 *R-tree Search Algorithm*

```
void RTreeSearch(Node* n, MBB* query)
{
    if (n->level > 0) /* this is an internal node in the tree */
    {
        for (int i=0; i< Number_Of_Child_Nodes; i++)
            if (MBBOverlap(query, n->child_mbb[i]))
            {
                RTreeSearch(n->child[i], query);
            }
    }
    else /* this is a leaf node */
    {
        for (int i=0; i< Number_Of_Child_Nodes; i++)
            if (MBBOverlap(query, n->child_mbb[i]))
            {   // Found overlapping data
                SaveOverlappingData(n->child[i]);
            }
    }
}
```

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 [26]. The traversal algorithms of R-Trees and its variants are 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 [4]. For high dimensional vector datasets, k-nearest neighbor (kNN) queries [1] are more com-

mon access patterns than orthogonal range queries, and it has been shown that the brute-force kNN search algorithms on GPU show good performance [6, 10].

Algorithm 2 *CUDA code for Braided Parallel MPES Algorithm*

```
__global__ void BraidedParallelMPES(
    MBB* node, MBB* query,
    int *totalHit, int numOfThreads, int numOfData)
{
  int tid = threadIdx.x; // thread ID
  int bid = blockIdx.x; // block ID

  for(int i=0; i<numOfData; i+=numOfThreads)
  {
      if( node[i+tid].boundary->contain(query[bid].boundary))
      {  // Hit
          saveResult(tid, node[i+tid]->data);
      }
  }
}
```

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 state-of-the-art 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 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 [17]. 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 3.4, we describe how the recursive tree traversal algorithm that needs back-tracking 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 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.

Algorithm 3 *CUDA code for Data-Parallel Partitioned MPES*

```
__global__ void DataParallelPartitionedMPES(
    MBB* node, MBB* query, int *totalHit,
    int numOfThreads, int numOfBlocks, int numOfData)
{
  int tid = threadIdx.x; // thread ID
  int bid = blockIdx.x; // block ID

  int start = (numOfdata/numOfBlocks)*bid;
  int finish = (numOfdata/numOfBlocks)*(bid+1);

  for(int i=start; i<finish; i+=numOfThreads)
  {
      if( node[i+tid].boundary->contain(query.boundary))
      {  // Hit
          saveResult(tid, node[i+tid]->data);
      }
  }
}
```

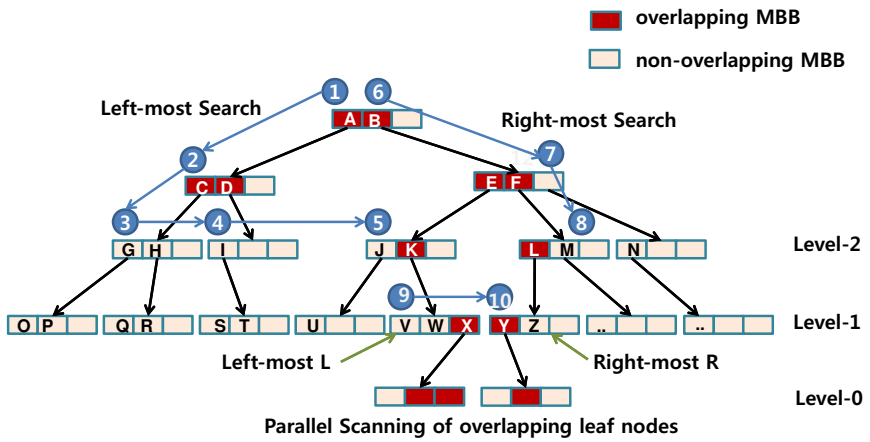


Fig. 3. *MPTS R-Tree Search with Sibling Check*

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

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 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 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 index is large and query range is also large. [1]

In order 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, MPTS search algorithm keeps choosing the leftmost node in each level in the first phase, (from step 1 to step 3), and in the second phase, the rightmost node in each level is visited (from step 4 to 6). Any node that is not in between the leftmost and rightmost nodes has no chance of overlapping the query. If there's an overlapping node outside 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 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 a MBB of child leaf node that overlaps the query, the child leaf node is fetched and the data stored in the leaf node are compared against the query.

In the example shown in Figure 3, let's assume a single warp consists of three threads and the maximum child nodes of each tree node is 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

[1] In 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*

```
__device__ long devMPTSRtreeFindLevelOneNode(Node *root, MBB *query, int LRflag)
{
  int tid = threadIdx.x; // thread id
  __shared__ boolean childOverlap[Number_Of_Child_Nodes];
  __shared__ Node* sn;
  sn = root;

  while(sn != NULL)
  {
      if (sn->level > 1)
      { // internal node
          // determine if "tid"th child overlaps the query
          childOverlap[tid] = 0;
          if (sn->child[tid].child && MBBOverlap(query,&sn->child[tid].mbb))
             // if child[tid] exists and the MBB of child[tid] overlaps a query
              childOverlap[tid] = tid;
          else
              childOverlap[tid] = LRFlag==LEFT? Number_Of_Child_Nodes + 1 : -1;
          __syncthreads();

          int N = Number_Of_Child_Nodes/2 + Number_Of_Child_Nodes%2;
          while(N > 1)
          {
              if(tid < N )
              {
                  // parallel reduction to find out leftmost/rightmost child
              }
              N = N/2+N%2;
              __syncthreads();
          }
          // childOverlap[0] holds the index of leftmost/rightmost child

          if( tid == 0)
          {
              if(none of the branches overlaps)
              {
                  sn = NULL;
                  if(LRFlag == LEFT && there is a right sibling )
                      sn = (Node*) ((char*) sn + TREENODESIZE);
                  if(LRFlag == RIGHT && there is a left sibling )
                      sn = (Node*) ((char*) sn - TREENODESIZE);
              }
              else
              {
                  // fetch the leftmost/rightmost child node
                  sn = sn->branch[ childOverlap[0] ].child;
              }
          }
          __syncthreads();
      }
      else  // this is a level-1 node (a parent of leaf node)
          return (long) sn;
  } // end of while
}
```

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

Algorithm 5 *Braided Parallel MPTS R-Trees Algorithm*

```
_global__ void BraidedParallelMPTSRTreeRangeQuery(Node* root, MBB* query)
{
   int bid = blockIdx.x; // block id
   int tid = threadIdx.x; // thread id

   // search leftmost and rightmost overlapping level-1 nodes
   Node* leftMost = devMPTSRtreeFindLevelOneNode(root, &query[bid], LEFT);
   Node* rightMost = devMPTSRtreeFindLevelOneNode(root, &query[bid], RIGHT);

   if(leftMost == NULL || rightMost == NULL) return;

   while(leftMost <= rightMost)
   {
       // fetch the next level-1 node and filter it out.

       for(int i=0;i<leftMost->nchild;i++)
       {
           Node* leaf = leftMost->child[i];
           if(MBBOverlap(query[bid],&leaf->child[tid].mbb))
           {   // Hit
               saveResult(tid, leaf->child[tid].data);
           }
       }
       leftMost += TREENODESIZE;
   }
}
```

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, which fetches its right sibling node when there's no overlapping MBB in 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, 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 higher level of trees, the number of pruned out 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. As we will discuss in Section 4, 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 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 leftmost leaf node and rightmost leaf node, we employed *Hilbert space filling curve* [20] 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.

3.5 Multi-threaded R-trees Search on 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* [20] as a de-clustering method so that each core gets similar amount of load for any shape of range query and we can maximize parallelism. However in our experimental study not shown in the paper, the simple round-robin assignment performed almost equally well with a Hilbert space filling curve. The experimental results presented in section 4 used round-robin selection.

4 Experiments

4.1 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* [25] 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.

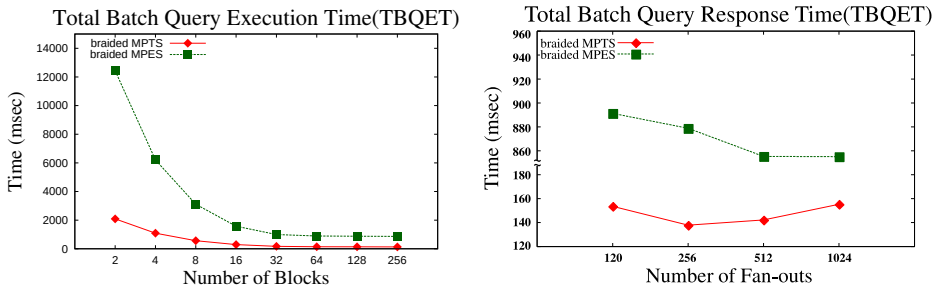
As for the performance metrics, we measure *TBQET (Total Batch Query Execution Time)* for query processing throughput and *QET (Query Execution Time)* for query response time. TBQET is the elapsed time between the time when the first query in a batch was submitted and the time when the last query in 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 GPU and CPU. With such ad-hoc optimizations 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.

4.2 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 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 4(a). When the number of blocks is greater than 128, the search time improvement seems minimal or sometimes we observe that the performance gets worse.

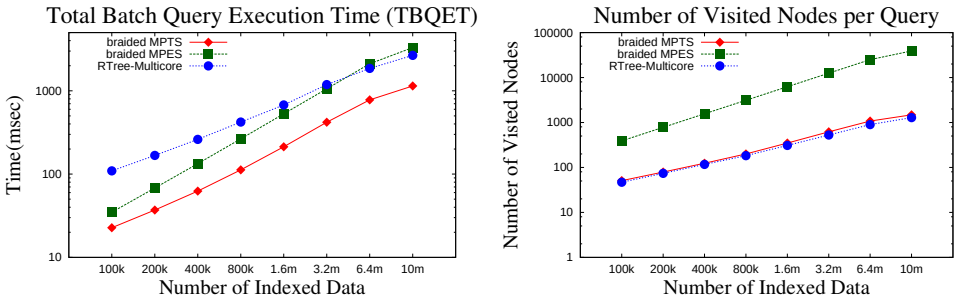
For the experiments shown in Figure 4(b), we increase the number of threads. Note that the number of threads in each block is equal to the number of node fan-outs for the MPTS R-trees as we discussed in section 3.4. The search performance of MPES improves as we increase the number of threads up to 512, 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



(a) Number of blocks (b) Number of threads per block

Fig. 4. *Throughput with Varying Number of Blocks and Threads per Block*

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 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.



(a) Query processing throughput (b) Number of visited nodes

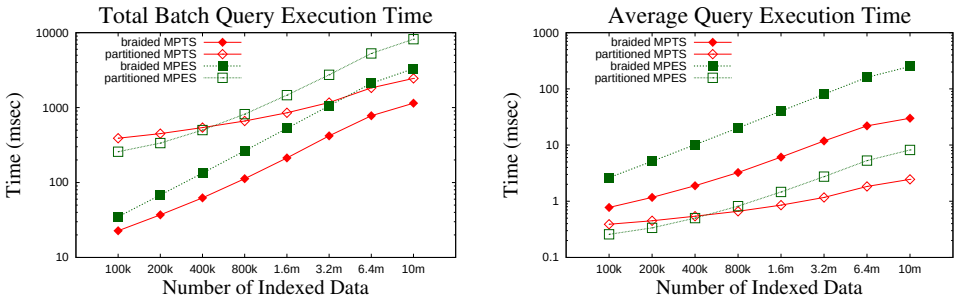
Fig. 5. *Effect of the Number of Indexed Data*

In Figure 5(a), 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 ratio [2] 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

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

Figure 5(a) 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 5(b) shows that the number of visited nodes of MPTS R-trees is about 1030 $\sim$ 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 GPU.



(a) Throughput with varying number of indexed data

(b) Query execution time with varying number of indexed data

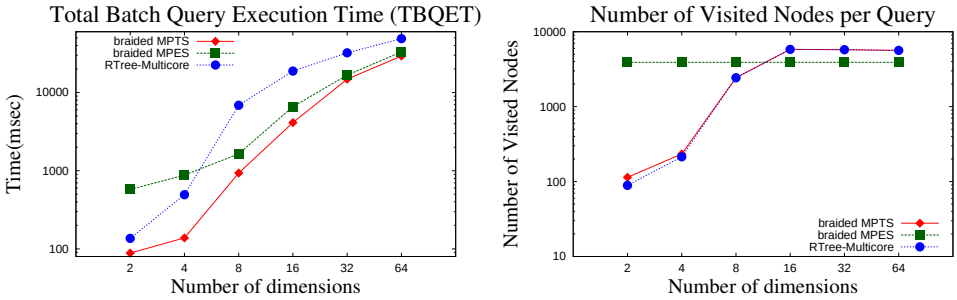
Fig. 6. *Braided Parallel Processing vs Data-parallel Partitioned Indexing*

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

Figure 6(a) 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 6(b), 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

dataset is very large. Although the braided parallel query processing improves the system throughput as shown in Figure 6(a), we show from this set of experiments that it is not effective in improving query response time.



(a) Query processing throughput (b) Number of visited nodes

Fig. 7. *Throughput and number of visited nodes high dimensions*

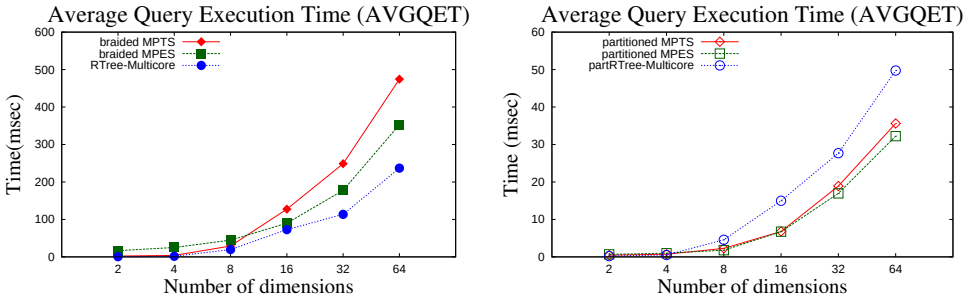
Figure 7 and 8 show the performance of index search for various dimensions, from two to sixty-four dimensions. For the experiments in high dimensions, we generate 1 million uniformly distributed hyper-cube data varying the number of dimensions, and submit 1,000 synthetically generated range queries in 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 3.3, it has been well known that brute-force linear scanning works faster than sophisticatedly designed tree structured indexes when the datasets are in high-dimensional space [5]. 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 7(b) 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 memory block is empty in MPTS R-trees and the number of instructions executed by a warp is fewer.

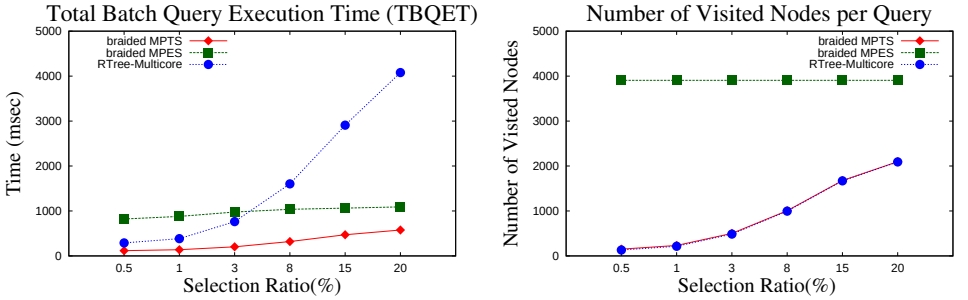


(a) Braided parallel indexing (b) Data-parallel partitioned indexing

Fig. 8. *Query execution time in high dimensions*

Figure 8 shows average query response time for individual queries. The data-parallel partitioned indexing schemes shown in Figure 8(b) processes individual queries more than 100 times faster than braided parallel indexing shown in Figure 8(a), In Figure 8(a), 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 GPU outperforms both braided-parallel and data-parallel partitioned R-Trees indexing on multi-core CPUs (RTree-Multicore and partRTree-Multicore).

Figure 9 and 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.



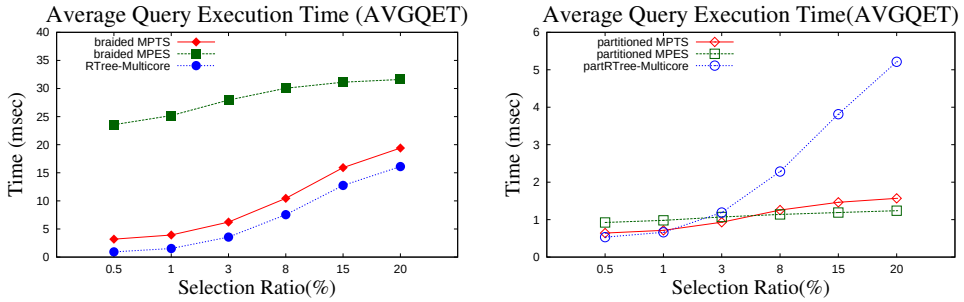
(a) Query processing throughput (b) Number of visited nodes

Fig. 9. *Throughput and number of visited nodes with varying selection ratio*

In the experiments shown in Figure 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 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.

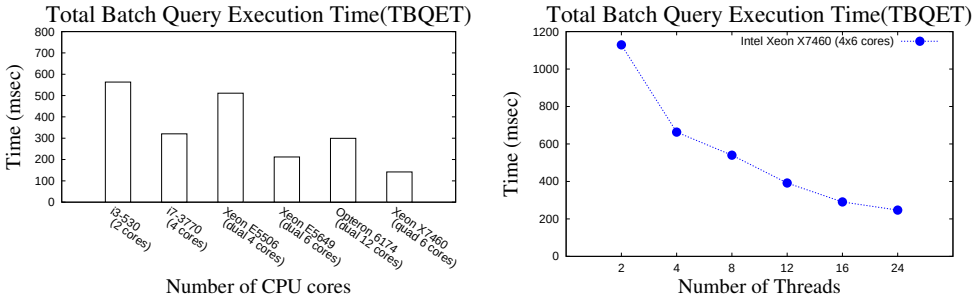


(a) Braided parallel indexing (b) Data-parallel partitioned indexing

Fig. 10. *Query execution time with varying selection ratio*

Figure 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 11(a), 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

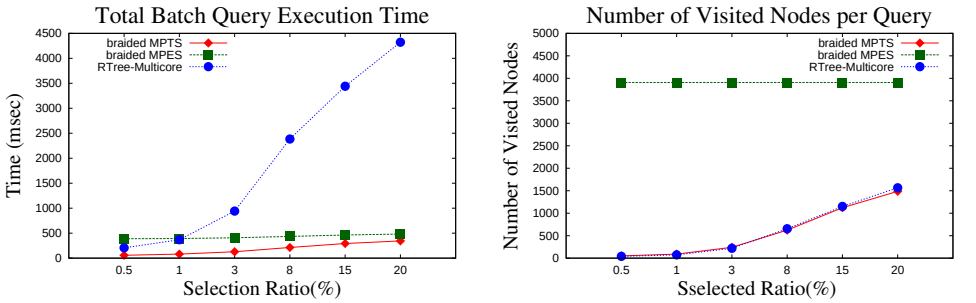


(a) Throughput on various CPUs

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

Fig. 11. *Performance evaluation on various multi-core architectures*

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 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 11(b), 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.



(a) Query processing throughput

(b) Number of visited nodes

Fig. 12. *Performance evaluation using San Francisco roadmap data*

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 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 12(b), the number of visited node with MPES is about 3 times greater than that of RTree-Multicore, but MPES takes the advantage 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.

5 Conclusions

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 GPU. The search time of MPTS algorithm on the state-of-the-art 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.

While we believe we have made progress in multi-dimensional indexing on GPU, we intend to extend this work for other types of multi-dimensional queries as well, such as nearest neighbor queries. The nearest neighbor query algorithms designed so far for GPUs are based on brute-force scanning. In the near future, we plan to convert recursive nearest neighbor query algorithms of R-trees into sequential multiple-phase scanning algorithm for GPU.

6 Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2011-001475) and KEIT of Korea funded by the IT R&D program MKE/KEIT (No. 10041608).

Jinwoong Kim and Beomseok Nam equally contributed to the design, implementation, performance study of the algorithm, and preparation of the manuscript. Sul-Gi Kim helped Jinwoong with minor code modification and proofread the manuscript. We sincerely thank professor Won-Ki Jeong for helping us analyze the performance on the GPU. We also would like to express sincere gratitude to the anonymous reviewers for their deep insights into the problems and valuable suggestions.

References

[1] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[2] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[3] Norbery Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 322–331, May 1990.

[4] Richard E. Bellman. *Adaptive Control Processes: A GuidedTour*. Princeton University Press, NJ, 1961.

[5] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, pages 28–39, 1996.

[6] Lawrence Cayton. Accelerating nearest neighbor search on manycore systems. In *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[7] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 440–447, 1999.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[9] Jordan Fix, Andrew Wilkes, and Kevin 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.

[10] Vincent Garcia and Frank 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.

[11] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2004.

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

[13] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query co-processing on graphics processors. volume 34, 2009.

[14] Tim Kaldewey, Jeff Hagen, Andrea Di Blas, and Eric Sedlar. Parallel search on video cards. In *Proceedings of the First USENIX conference on Hot topics in parallelism (HotPar 09)*, 2009.

[15] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 195–204, 1992.

[16] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep 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.

[17] Jinwoong Kim, Sumin Hong, and Beomseok 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.

[18] Nick Koudas, Christos Faloutsos, and Ibrahim Kamel. Declustering spatial databases on a multi-computer architecture. In *Proceedings of the 5th International Conference on Extending Databases Technology (EDBT)*, 1996.

[19] Lijuan Luo, Martin D.F. Wong, and Lance Leong. Parallel implementation of R-trees on the GPU. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

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

[21] Beomseok Nam and Alan 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.

[22] Beomseok Nam, Minho Shin, Henrique Andrade, and Alan Sussman. Multiple query scheduling for distributed semantic caches. *Journal of Parallel and Distributed Computing*, 70(5):598–611, 2010.

[23] Beomseok Nam and Alan Sussman. Analyzing design choices for distributed multidimensional indexing. *Journal of Supercomputing*, 59(3):1552–1576, 2012.

[24] Bernd Schnitzer and Scott 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)*, pages 68–77, 1999.

[25] Yannis Theodoridis. R-tree Portal. *http://www.rtreeportal.org*.

[26] Roger Weber, Hans-J. Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[27] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.