2012 IEEE 14th International Conference on High Performance Computing and Communications

A Performance Study of Traversing Spatial Indexing Structures in Parallel on GPU

Jinwoong Kim, Sumin Hong, Beomseok Nam
School of Electrical and Computer Engineering,
Ulsan National Institute of Science and Technology,
Ulsan, Republic of Korea, 689-798
Email: {jwkim,sumin246,bsnam}@unist.ac.kr

***Abstract*—CUDA is a parallel programming environment that enables significant performance improvement by leveraging the massively parallel processing capability of the GPU. Inherently spatial indexing structures such as R-Trees are not well suited for CUDA environment due to its irregular tree traversal for range queries. Traversing irregular tree search paths makes it hard to maximize the utilization of many-core architectures. In this paper, we propose assigning an individual sub-tree to each SMP (streaming multi-processor) in GPGPU, such that CUDA cores in the same SMP co-operate to navigate tree index nodes. This parallel partitioned-indexing improves the utilization of many cores in GPGPU significantly. Also, we propose a new range query search algorithm - *three-phase-search* that avoids non-sequential random access to tree nodes and accelerates the search performance of spatial indexing structures on GPU. Our experimental results show that GPU-based parallel spatial indexing scheme on NVIDA Tesla M2090 GPGPU outperforms the CPU-based multi-threaded R-trees on AMD Opteron 6128HE processor by two times.**

***Keywords*-Multi-dimensional indexing; Multi-dimensional range query; GPGPU indexing;**

I. Introduction

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. Multidimensional range query is one of the most common access patterns into such datasets, and is an important class of problems in data-intensive scientific computing and computer graphics as well. In order to handle multidimensional range queries efficiently, a large number of efficient and scalable indexing structures (e.g. R-trees [1]) have been proposed and improved.

Recently GPGPU has emerged successfully as a new parallel computing paradigm, and is being used to accelerate many general-purpose computation in various fields. Until recently, GPU had very limited programming functionality and was used only for computer graphics computation. Although NVIDIA keeps improving CUDA architecture and programming model that allows a programmer to write general-purpose parallel programs on GPU, still GPGPU has many restrictions and it's not trivial to convert various single threaded algorithms into parallel shared memory algorithms. However, it draws lots of attention due to its significant speedups over single threaded program.

Traversing spatial indexing structures on GPU is not an easy problem due to its irregular tree search paths. With a given internal tree node and a range query, more than one child nodes may overlap the given query range and the sub-trees of all the overlapping child nodes have to be visited recursively. For the range queries, the number of overlapping child nodes is not known priori, hence the traditional search algorithms of spatial indexing structures use recursion to visit all the tree nodes for the irregular search paths. However as we will discuss in the experimental section, we found out the recursive functions do not perform well in CUDA architecture.

In CUDA architecture, the index search performance speed-up 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, many cores in GPU can not be well utilized and it is difficult to navigate spatial indexing structures in parallel.

In this paper, we propose partitioning a spatial indexing structure into small sub-trees and assigning an individual sub-tree to each SMP (streaming multi-processor) in GPGPU. In this parallel scheme, all the CUDA cores in the same SMP co-operate to process the assigned tree index node in parallel. We also propose a *three-phase-search algorithm* for a spatial indexing structure - KDB-tree [2], which enables us to avoid irregular search path. To the best of our knowledge, this paper presents the first CUDA-aware range query algorithm for multi-dimensional indexing.

The rest of the paper is organized as follows. In Section II we discuss other works related to the indexing on GPU. In Section III we propose three CUDA-aware indexing schemes, and discuss experimental results in Section IV. In Section V we conclude and discuss future work.

II. Related Works

In spatio-temporal database community, there has been extensive research on multi-dimensional indexing structures, starting with the seminal work on R-trees [1]. There were also some efforts to parallelize the R-trees in shared-nothing

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2.120089.01).

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.121

environment. Kamel and Faloutsos proposed Multiplexed R-trees [3], for a machine with a single CPU and multiple disks. Koudas et al. proposed a Master R-trees [4], and Master Client R-trees was proposed by Schnitzer et al. [5] for distributed parallel cluster machines.

As multi-core architectures have evolved, a couple of recent efforts were made to exploit SIMD execution of GPGPU to improve database query performance. Kaldewey et al. [6] 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 [7]. Each block of FAST is the unit of parallel processing in a single streaming multiprocessor (SMP) of GPU. The block is similar to the node of R-trees in a sense that its size is chosen to avoid the bandwidth bottleneck between main memory and GPU device memory.

Although GPUs have a large number of cores, each core 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 to reduce response time of each query. Fix et al. proposed *braided parallelism* for traversing B+-Trees, wherein a single B+-Trees exists in GPU memory and multiple independent queries are concurrently executed within each GPU core [8]. They showed this approach yields higher query processing throughput, but it does not help improve individual query response time.

In addition to exploiting GPU's massive parallelism for indexing data structures, SQL processing with GPU has been also attempted by [9], [10], [11]. Bakkum et al. have implemented SQLite database virtual machine on GPU, and improved the performance of some types of SELECT queries [11]. Che et al. [9] 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 [10]. Govindaraju et al. [12] also proposed GPU-aware algorithms for several common database operations such as conjunctive selections, aggregations, and semi-linear queries.

III. CUDA-ENABLED PARALLEL SPATIAL INDEXING STRUCTURES

A. 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 need to share intermediate data results and cooperate on memory access with each other through thread 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.

Algorithm 1 *CUDA code for massively parallel exhaustive scanning (MPES)*

```
__global__ void linearScanningKernel(
    struct Rect* node, struct Rect query,
    int *totalHit, int numOfthreads, int numOfdata)
{
  int tid = threadIdx.x; // thread ID
  int bid = blockIdx.x; // block ID
  int pos = (numOfdata/numOfThreads)*id;

  int hit = 0;
  for(int i=0; i<numOfdata/numOfThreads; i++) {
    if( node[pos+i].boundary->contain(query.boundary)){
      hit++; //increase each thread's hit count
      storeInGlobalMemory(id, node[pos+i]);
    }
  }

  totalHit[id] = hit;
}
```

A CUDA kernel function can specify the number of blocks and the number of threads per block. The blocks are distributed across the multiple SMPs, and each SMP executes a block of threads in parallel. However, the number of concurrent threads 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.

B. Multi-threaded Parallel R-trees Search (MTPR) on CPU Multi-cores

GPU is not the only hardware component that exploits parallelism in modern computer architectures. Multi-core processors, like *Nehalem* or *Opteron* are common these days, and many scientific applications are easily implemented on them to exploit the multi-core architectures.

In order to accelerate the access to R-trees [1] index on multi-core processors, we partitioned a single R-trees index into multiple small partitioned-trees and assigned each of them to a single thread, i.e. N threads will concurrently search N number of partitioned-trees. In this multi-threaded parallel R-trees search scheme (MTPR), the number of partitioned-trees is determined by the total number of threads.

This approach allows us to search small independent indexes concurrently. When inserting a new data, we chose a partitioned-tree to insert in a round-robin fashion. Alternatively we can employ *Hilbert space filling curve* [13] which is known as a good spatio-temporal declustering method so

Algorithm 2 *CUDA code for massively parallel CUDA R-tree search (MPCR)*

```
__global__ void MPCRSearchKernel(Rect query, int* hit) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  // Each thread searches its own R-tree
  hit[tid] = deviceMPCRSearch(subTree[tid], &query);
}

__device__ int deviceMPCRSearch(Node *node, Rect *query) {
  int i, hitCount = 0;

  if (node->level > 0)  // an internal node
    for (i=0; i<node->numChild; i++){
      // copy from global memory
      Node c = *(node->child[i]);

      // this branch overlaps the query
      if( query->overlap(c.mbb))
          hitCount += deviceRTreeSearch(&c, query);
    }
  else // a leaf node
    for (i=0; i<node->numChild; i++) {
      // copy from global memory
      Node c = *(node->child[i]);

      // this branch overlaps the query
      if( query->overlap(c.mbb)){
          // this child overlaps the query
          saveResult(c.data); // Hit
          hitCount++;
      }
    }
  return hitCount;
}
```

that each core gets similar amount of load for any shape of range queries and maximize parallelism. However in our experimental study, the simple round-robin assignment performed almost equally well with a Hilbert space filling curve.

C. Massively Parallel Exhaustive Scanning (MPES) on GPU

Although GPU programming model has been improved for general purpose applications, GPUs are still very restrictive and need to be improved in terms of programming complexity and performance. Since GPUs can process many data items in parallel, GPUs seem to be only effective for computer graphics and streaming data processing. Hence, if possible, often it is desirable to modify random data access pattern into sequential stream processing in order to get the maximum performance out of CUDA architecture. As we described earlier, the tree node traversal pattern is inherently irregular and recursive function call on 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 in order to transform range query search into a streaming data filtering process, and to exploit SIMD execution model of CUDA architecture. The massively parallel exhaustive scanning (MPES) algorithm is shown in Algorithm 1. 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

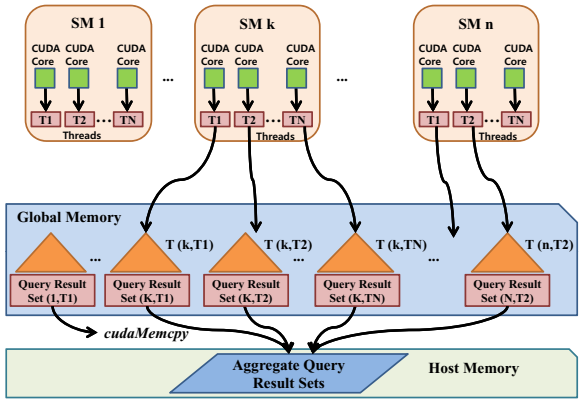

Figure 1: *Massively Parallel CUDA R-Trees (MPCR)*

the assigned dataset overlaps the given query.

D. Massively Parallel CUDA R-trees Search (MPCR) on GPU Many-cores

In *massively parallel CUDA R-trees (MPCR)*, we partition a single R-Trees into N small partitioned-trees as in multi-threaded parallel R-trees search (MTPR). In the MPCR shown in Figure 1, the number of partitioned-trees is set to the number of CUDA threads on GPU, i.e. each SMP searches multiple small partitioned-trees concurrently. The search kernel function of MPCR is shown in Algorithm 2.

In MPCR implementation, a new data is inserted into one of the partitioned-trees in host memory in a round-robin fashion, and the R-Trees are copied to GPU global memory. Due to its non-parallel nature and the complexity of node insertion and split algorithms, R-trees insertion operation doesn't perform well on GPU. Hence we implemented a CUDA kernel function that copies R-trees from host memory to GPU global memory. Although CUDA memory copy function is an expensive operation, we observed that memory copy from host to CUDA global memory takes less amount of time than inserting a new data directly into GPU global memory.

When a client submits a query, CUDA threads execute the same kernel search function with their own partitioned-trees concurrently. Due to the limited size of local and shared memory space of CUDA architectures (48 KB), we could not store all the partitioned-trees in shared memory but in larger global memory. Using the block index and thread index, each thread identifies which partitioned-trees to search. Since latest CUDA programming model supports recursive function call, we implemented the search function as a recursive device function as in traditional R-tree search algorithm. For each recursion, each thread needs to fetch a child node and access global memory, which turned out to be a performance bottleneck in MPCR. 16 threads within the same SMP compete for global memory access and

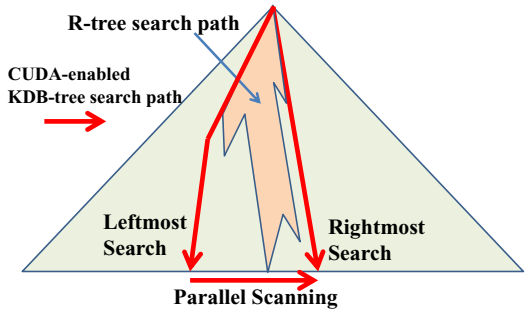

Figure 2: *Three-phase KDB-Tree Search (3P-KDB)*

they also compete with the threads in other SMP since the global memory is shared by all the SMPs. After all the threads finish searching partitioned-trees, the query results are collected, merged, and returned back to a client.

E. Three-Phase Parallel KDB-tree (3P-KDB) Search on GPU

As shown in Algorithm 2, traversing R-Trees requires 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, we have implemented an alternative approach to parallelize search operation of each tree internal nodes. In *three-phase parallel KDB-tree search scheme (3P-KDB)*, we allocate a CUDA thread block to each partitioned-tree, and a set of threads in a block cooperate to search the MBBs of child nodes in parallel. Thus, the number of partitioned-trees is set equal to the number of thread blocks, and the number of threads in each block is set to the node fan-outs (the maximum number of child nodes in tree structures).

In this block-based parallel search scheme, all the threads in a single block read the same tree node and each of them independently checks whether each child node of the current tree node overlaps a given range query. If there are more than one child nodes that overlap the given query, traditional spatial index search algorithm visits all of them recursively. However we observed that the recursion performs really bad in CUDA architecture. Instead of recursive search function, we tried to employ a shared queue where the threads insert the overlapping child nodes. When all threads are done with processing a current tree node, they fetch the next tree node from the queue, compare the MBB of child nodes again in parallel, insert the overlapping child nodes into the queue, and repeat. The problem of this approach is that the size of queue can grow almost as big as the whole tree. Thus the queue must reside in the global memory and the threads compete for global memory access, which makes the global memory access a performance bottleneck again as in MPCR.

In order to avoid the irregular tree traversal and to avoid both recursion and global queue access, we designed *three-phase KDB-tree search algorithm (3P-KDB)*. The three-phase KDB-tree search algorithm always selects only one child node no matter how many child nodes overlap the

Algorithm 3 *CUDA code for Three-phase KDB-tree (3P-KDB) search*

```
__global__ void TPSearchKernel(Rect q) {
  int bid = blockIdx.x; // block id
  int tid = threadIdx.x; // thread id

  // Each block searches its own partTree[bid] in parallel.

  // search leftmost and rightmost overlapping leaf nodes
  long leftMost = dTPSearch(partTree[bid], &q, tid, MIN);
  long rightMost = dTPSearch(partTree[bid], &q, tid, MAX);

  while(leftMost <= rightMost){
      // fetch the next leaf node and filter it out.
      dTPSearchLeaf( (Node*) leftMost, &q, tid);
      leftMost += PGSIZE;
  }
}

__device__ long dTPSearch(Node *n, Rect *q,
                          int tid, int flag)
{
  __shared__ int childOverlap[NODECARD];
  __shared__ Node* sn;
  sn = n;  // n is the root of partitioned-tree

  while(1){
    if (sn->level > 0){
      // sn is an internal node in the tree

      // determine if "tid"th child overlaps the query
      childOverlap[tid] = 0;
      if (sn->child[tid].child &&
          dRectOverlap(q,&sn->child[tid].rect))
                childOverlap[tid] = tid;
      else childOverlap[tid] = -1;
      __syncthreads();

      // reduction to find out leftmost/rightmost child
      int N = 1024 / 2;
      while(N > 0){
          if(tid < N && (tid+N) < NODECARD ) {
              if(flag==MAX &&  // rightmost
                 childOverlap[tid] < childOverlap[tid+N] )
                  childOverlap[tid] = childOverlap[tid+N];
              if(flag==MIN &&  // leftmost
                 childOverlap[tid] > childOverlap[tid+N] )
                  childOverlap[tid] = childOverlap[tid+N];
          }
          N = N/2;
          __syncthreads();
      }

      if( tid == 0)
          sn = sn->branch[ childOverlap[0] ].child;

      __syncthreads();
    }
    else  // this is a leaf node
      return (long) sn;
  } // end of while
}

__device__ void dTPSearchLeaf(Node *leaf, Rect *q, int tid) {
  if (leaf->child[tid].child &&
      dRectOverlap(r,&leaf->child[tid].rect)) {
          saveResult(leaf->child[tid].data); // Hit
  }
}
```

given query. With a given range query, the spatial indexing search path is irregular by its nature as illustrated in Figure 2. While navigating down the tree nodes, it may fail finding an overlapping child nodes, return back to its parent, and search its siblings. This causes a large number of back-tracking and random access to global memory. KDB-Trees [2] is different from R-Trees in a sense that KDB-Trees always reaches at least one child node even if the leaf node doesn't contain any overlapping data. But KDB-Trees also perform back-tracking to search other leaf nodes whose MBB overlap the given range query.

3P-KDB search scheme avoids the expensive back-tracking by choosing only a single leftmost node from root to leaf level in the first phase, and a single rightmost node from root to leaf level in the second phase. In the last phase, 3P-KDB search scans all the leaf nodes in the middle. Algorithm 3 shows how to identify the leftmost or rightmost child node when traversing internal tree nodes. We perform CUDA-style parallel reduction algorithm to identify which child is overlapping and located in the leftmost position in a tree hierarchy. The first and second phase can run concurrently in a separate group of threads for further optimization. But the overhead of the first and second phase is very small since the number of internal nodes that need to be read from global memory is always just $2 \times log(N)$, $log(N)$ for the leftmost leaf and another $log(N)$ for the rightmost leaf, where the N is the number of indexed data.

Since we select at least one and only one child node at a time as we navigate down the tree, the back-tracking or global queue/stack is not necessary. As we search down one level in the tree structure, we need to read just one leftmost (or one rightmost) tree node from global memory. Without visiting the overlapping nodes in the middle, we can avoid expensive recursion and random global memory access. But the price to pay for that is we have to visit all the leaf nodes in between the leftmost leaf and the rightmost leaf. Fortunately, CUDA is known for its outstanding performance of processing sequential data. When we transfer the indexing structure from host to GPU global memory, the tree nodes are ordered in a breadth-first manner. Thus we can simply fetch the next sibling leaf node by adding the node size to the current leaf node's memory address.

IV. Experiments

We measured search performance of the proposed parallel spatial indexing schemes on a machine running Ubuntu Linux 10.1 with CUDA Toolkit 3.2. The host machine is equipped with AMD Opteron 8 Core 6128HE 2.0GHz processor and 64GB DDR3 memory. The GPU video card that we used was Tesla Fermi M2090 GPU card. It has 16 SMPs and each SMP has 32 CUDA cores, which enables 512 (16x32) threads to run concurrently. To evaluate the proposed parallel index search schemes, we generated 10 million two-dimensional point data following uniform, normal, and Zipf's distribution, but we present the experimental results of only the uniform distribution since the results of the other distributions are similar.

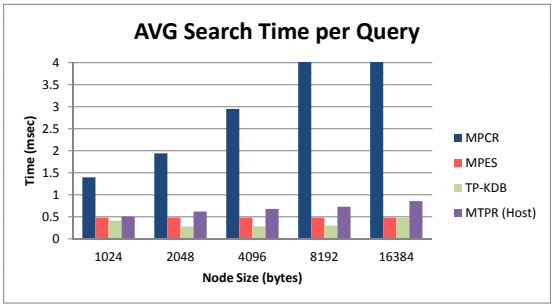

Figure 3: *Average Query Response Time with Varying Page Size*

In Figure 3, we measured the average query response time with various tree node size. Note that the tree node size is determined by disk page size for disk-based indexing structures. But for the memory based indexing structures the node size doesn't have to be the same with physical disk page size. CUDA programming model doesn't allow us to directly access disk pages, thus the node size should be determined considering the number of CUDA threads in each block.

Unlike TP-KDB, the search time of MPCR becomes worse as we increase the tree node size. As we have a larger tree node size, MPCR makes each GPU core compare more number of MBBs with a given range query without getting help from more data parallelism, but TP-KDB spawns more number of threads for the increased number of MBBs and it improves the search performance. The TP-KDB shows the fastest search performance when the node size is 8 KB and the number of threads is 340. When the node size is larger than that, i.e. 16 KB, 680 CUDA threads compete for only 32 available CUDA cores and it seems to hurt the search performance. Note that changing the node size doesn't affect the search time of MPES because the data are stored in a sequential array in MPES and the amount of work to be performed is determined by the total number of data, not by the node size. Interestingly MPES is much faster than MTPR on CPU and MPCR on GPU.

Figure 4 shows the average query response time as we increase the number of thread blocks. For this set of experiments, we fixed the number of threads per block to 340. Accordingly the tree node size of TP-KDB was fixed to 8 KB so that 340 threads in a block can compare the MBB of each of 340 child nodes. For MPCR, we set the tree node size to 256 bytes since a larger node size hurts the search performance.

As we increase the number of blocks from 1 to 128, the

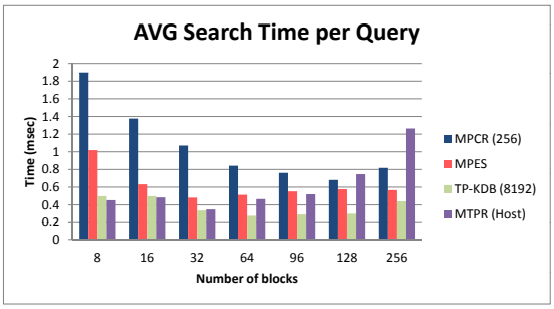

Figure 4: *Average Query Response Time with Varying Number of Blocks*

query response time keeps decreasing sub-linearly for all the parallel spatial indexing schemes since we exploit more parallelism. However as we increase the number of blocks larger than 128, a larger number of blocks start suffering from the resource contention and switching overhead, resulting in decreased search performance. In Tesla M2090, the maximum number of *resident blocks* per SMP is 8 and the number of SMPs is 16. Thus the best performance is observed when the number of blocks is 128 (8×16) due to the maximum core utilization and minimal switching overhead. As we increase or decrease the number of blocks, it suffers from switching overhead or low core utilization.

Throughout the experiments with various number of blocks, TP-KDB search scheme consistently outperforms both MPES and MPCR. The average search time of TP-KDB is only 26∼53% of MPCR and 48∼78% of MPES. The MTPR on CPU multi-cores shows good performance up to 32 partitioned-trees, but as we add more number of threads, its search performance degrades as AMD Opteron processor has only 8 cores. When the number of blocks is small, MPCR shows even worse performance than MPES and MTPR. This is mainly because MPCR was not designed considering the architectural characteristics of GPU, which contains a large number of parallel but low performing cores. Presently, the computation power of a single core in AMD Opteron processor is much higher than that of CUDA core. We measured the R-tree search time on CPU and GPU using a single thread, and the search time on CPU was only 18% of the search time on GPU. As we increase the number of threads and the number of partitioned-trees, MPCR on GPU takes the advantage of more available GPU cores but still the average search time of MPCR is higher than that of MTPR on CPU until the number of blocks is less than 96.

V. Conclusions

In this work, we propose CUDA-based parallel spatial indexing schemes and a 3-phase search algorithm. The proposed schemes improve the utilization of many-core architecture for spatial indexing, and avoid the irregular search path by converting the tree traversal problem into a sequential data processing problem. Our experimental results show that the search time using three-phase search algorithm on GPU is as low as 48% of parallel R-trees on multi-core CPU architecture. As a future direction of this work, we plan to optimize the order of child nodes in order to make the search path narrow and access less number of leaf nodes in global memory.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query co-processing on graphics processors," vol. 34, no. 4, 2009.

[11] 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.

[12] 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.

[13] 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.