A Parallel Access Method for Spatial Data Using GPU

Byoung-Woo Oh
Department of Computer Engineering
Kumoh National Institute of Technology
Gumi, Korea
bwoh@kumoh.ac.kr

***Abstract*— Spatial access methods (SAMs) are used for information retrieval in large spatial databases. Many of the SAMs use sequential tree structures to search the result set of the spatial data which are contained in the given query region. In order to improve performance for the SAM, this paper proposes a parallel method using GPU. Since the searching process needs intensive computation but is independently examined on a lot of the MBRs of the spatial data, the spatial search function can be efficiently computed on GPU in a massive parallel way. The proposed method achieves high speed by efficiently utilizing the parallelism of the GPU throughout the whole process and by reducing the transfer latency between CPU and GPU with the memory structure which resides in GPU memory at all times and with the usage of bit-wise operation for the result set. In order to measure the speedup achieved by the proposed parallel method, the execution time is compared with the sequential R*-tree that is loaded in the main memory and executed on CPU.**

Keywords-spatial data; GPU; CUDA;parallel computing; spatial access method

I. INTRODUCTION

Recently, driven by the user demand for rapid 3D gaming environment, Graphic Processing Unit (GPU) has improved. GPU has been developed to not only improve the graphics capabilities but also support general purpose computations. As a result, there has been a growing trend of using GPU for general-purpose parallel computing. GPGPU stands for General-Purpose computation on GPU. GPUs are high-performance many-core processors capable of very high computation and data throughput [1]. CUDA™ by NVIDIA is a general purpose and data-parallel computing architecture [2]. The data-parallel computing provides efficient processing with a same program and many data elements in parallel, so called Single-Instruction Multiple-Data (SIMD). Many applications that need computational power for complex calculation have studied to use GPGPU to enhance their performance [3, 4, 5, 6, 7, 8]. Considering the large amount spatial data, the techniques for processing spatial data can be one of the application.

In order to handle spatial data efficiently, tree structures have been commonly used such as R-tree and R*-tree [9, 10]. The tree structure provides searching for a set of spatial objects that are contained within a given query region. The search process sequentially examine nodes of a tree from root node to leaf node. Because the search process manipulates a large set of spatial data, it can exploits a data-parallel programming model to speed up the computations.

Instead of using the common sequential tree for the search process, this paper proposes a method that uses the data-parallel programming model to find a set of spatial objects that are contained in the given query region using GPU. In designing a parallel method using GPU, there are two challenging issues. The first is how to maximally exploit the GPU cores when searching a result set of spatial objects. CUDA introduces the kernel function which can be assigned to many threads and executed in parallel. The proposed method uses the CUDA kernel function to be executed in multiple threads. The kernel function examines whether the MBR of the corresponding spatial object is contained by the query region or not.

Another issue is how to reduce the data transfer between CPU main memory and GPU device memory. The MBR data of the spatial objects and the result data can be transferred between CPU and GPU. Because the MBR data are used whenever the searching process is executed, it is desirable that the MBR data reside in GPU memory while the application is running. In order to keep the MBR data in GPU, the initial loading process reads the spatial objects and transfers the MBR data to GPU when the application begins. After the searching process, the result data should be sent back to CPU to manipulate them. In order to reduce the transfer time, the size of the result data is minimized by using an array of bits that indicate whether the corresponding objects are selected or not.

In order to measure the speedup achieved by the proposed parallel method, the execution time is compared with the main memory based R*-tree. The experimental result shows that the proposed parallel method on GPU outperforms significantly the sequential R*-tree on CPU.

The rest of the paper is organized as follows. In section II, GPGPU and R*-tree are briefly described. Section III introduces the parallel access method using GPU. Section IV explains the experimental result. Finally, section V describes the concluding remarks and the future work.

II. Related Work

This chapter introduces the trend of parallel programming using graphic card and traditional sequential approach for searching spatial data.

A. General-Purpose Computing on Graphics Processing Units

Driven by the user demand for rapid 3D gaming environment, GPU has improved. The manufacturers of GPUs provide programming toolkit to use the computational power of GPUs for general-purpose computing. GPGPU provides highly parallel computation with multi-processor and multi-thread. Many applications that need computational power for complex calculation use GPGPU to enhance their performance. Especially, applications that process large data sets can speed up the computations because GPGPU supports data-parallel processing. The data-parallel processing means that the same program is executed on many data elements in parallel with multi-processor and multi-thread. The major manufacturers of GPUs are NVIDIA and AMD (ATI). NVIDIA and AMD introduced CUDA and AMD Stream respectively as software development kit for GPGPU [2, 11].

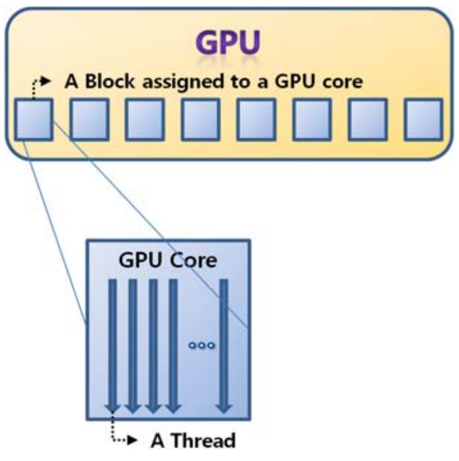
CUDA introduces the kernel function which can be assigned to many threads and executed in parallel. Any call to a kernel function must specify the number of blocks and the number of threads. Each thread that executes the kernel is allocated to a GPU core according to the given number of blocks and the given number of threads as shown in Fig. 1.

CUDA provides built-in variables such as threadIdx, blockDim and blockIdx. The threadIdx is an index to identify a thread in a block. The block is a set of threads. The blockDim is a number of threads that are forming a block. The blockIdx is an index to identify a block. The built-in variables can be identified by a one-dimensional, two-dimensional, or three-dimensional index and accessible within the kernel function. Fig. 1(b) shows an example of one-dimensional variables with x axis.

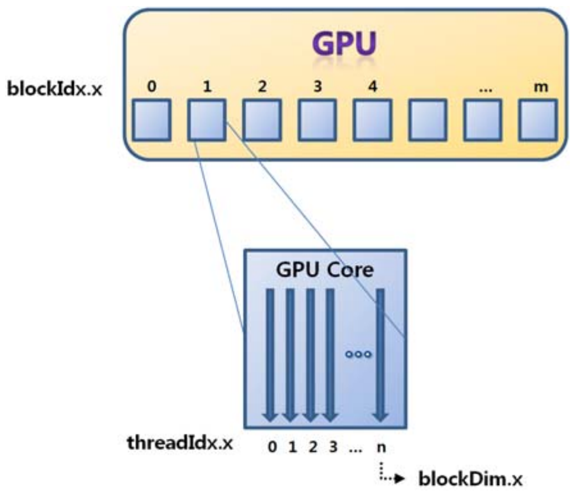
This paper proposed a parallel access method for spatial data using CUDA that provides SIMD (Single-Instruction Multiple-Data) with SIMT (Single-Instruction, Multiple-Tread) architecture. The CUDA kernel function is used to search spatial object within multiple threads and uses built-in variables to identify a thread.

B. Spatial Access Methods

Spatial Access Methods (SAMs) are used by spatial databases, that store data related to objects in space, to optimize spatial queries. The R*-tree is a well-known and traditional data structure for spatial data [10]. The geometries of spatial data are set of n-dimensional coordinates, e.g. polygons or poly-lines. They are too complex to check containment, intersection or overlap. The R*-tree uses technique to approximate complex spatial data to minimum bounding rectangles (MBRs) of whole coordinates. Because MBRs are simple rectangles that preserve the extension of the spatial object, the searching process can be simplified by comparing the MBR of query region and the MBRs of objects to find a result set of candidates. After finding a result set of candidates, the geometries of those candidates may be examined for the refinement.



(a) Block and thread



(b) Built-in variables for block and thread

Figure 1. CUDA programming model.

The R*-tree constructs a balanced tree structure and consists of root node, non-leaf node and leaf node. The leaf nodes contain pointers to data objects. The non-leaf nodes contain entries of a set of MBR and child-pointer that is the address of a lower node. The root node is a non-leaf node which located on the top of a tree and the starting node of spatial searching. The searching process descends sequentially from the root node to the leaf nodes in a recursive manner. Due to the hierarchical and recursive nature, the R*-tree is hard to run efficiently on SIMD hardware [4].

Instead of using the sequential and hierarchical tree, this paper proposes a method that uses the data-parallel programming model for efficient search. The proposed method is compared with the main memory based R*-tree to measure speedup for experimental result.

III. PARALLEL ACCESS METHOD FOR SPATIAL DATA

Since the searching process needs intensive computation but is independently examined on a lot of MBRs of spatial data, the spatial access method can be computed on GPU in a massive parallel way. The proposed parallel access method uses memory structures that can be divided into two groups by the stored location; CPU memory structures and GPU memory structures. The proposed method builds them in the initial loading process and exploits them in the searching process. This chapter describes the memory structures and the processes of the proposed parallel access method for spatial data.

A. Memory Structures

The proposed method uses data structures in both GPU and CPU memory as shown in Fig. 2. The GPU memory contains the MBR table and the result flag set. The CPU memory contains the copied result flag set from GPU, the mapping table and the geometry data.

The MBR table is an array that stores the minimum bounding rectangle of the spatial objects. An MBR is a maximum extents of a 2-dimensional spatial object and consists of four 32-bit integers; top, left, right and bottom value in a 2-dimesinal map. Because the number of MBR elements is the number of objects and the size of each element is 16 bytes, the size of the array is determined as N * 16 bytes when N is the number of objects. The order of an MBR element in the array is matched to the order of the corresponding spatial object that is stored in the geometry data in main memory. That is, the first MBR in the table stores the MBR of the first object in the geometry data. The MBR table is allocated and loaded when the geometry data are loaded. Once the MBR table is loaded, it resides in the global memory of the GPU to process spatial queries. Threads for search process check each MBR against the given query region in parallel.

The result flag set is an bit array that stores the result of a query. Because there is transfer latency of data communication between GPU and CPU, the size of the result flag set is an important factor for performance. To reduce the size and latency, a result of an object with a given query is assigned in a bit. If a bit that represents an object flag is set to 1, then the corresponding object is contained by the given query region. For example, the first bit of the second byte is set to 1 when the 9th object is found as a result. The size of the result flag set is (N + 7) / 8 bytes. The result flag set is allocated in the GPU memory and initialized as zero with the number of objects after the load of the geometry data. When a search is completed, the result flag set in the GPU should be transferred to

CPU. After the result flag set in the GPU is transferred to the CPU, it is set to zero for a next search instead of deleting it.

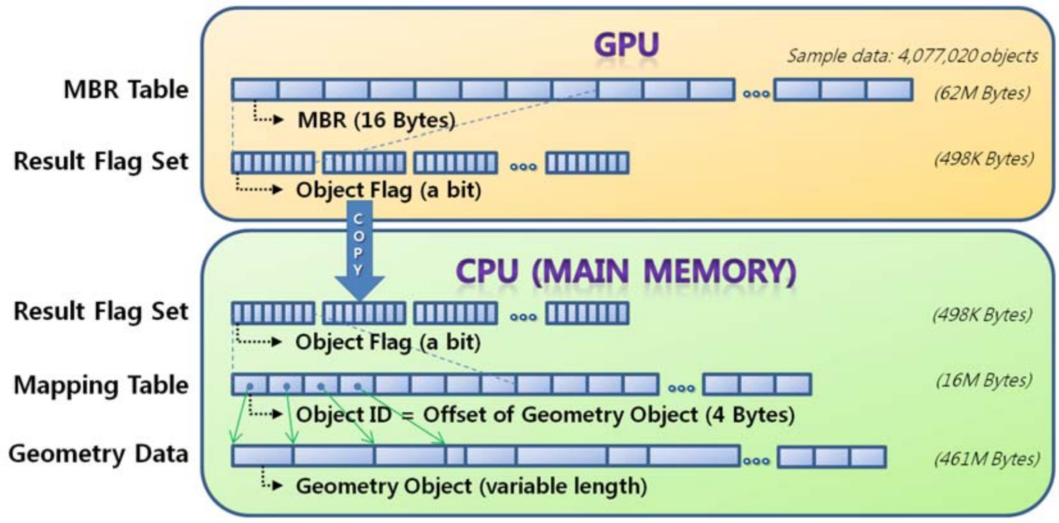


Figure 2. Memory structures in GPU and CPU.

The copied result flag set in the main memory for CPU is identical with that in the GPU. The only difference is the location where they are stored; the global memory in the GPU or the main memory for the CPU. The result flag set in the CPU can be used to access the found objects in the geometry data by referencing the mapping table.

The mapping table in the main memory for CPU is an array that can map an object index to a starting offset in the geometry data. It provides indirect referencing to the spatial objects. Because the length of geometry data is variable length, it is necessary to maintain starting offsets for accessing objects. An element of the mapping table contains an address to reference the corresponding object. An object ID is a unique identifier to distinguish objects for the R*-tree. The starting offset in the geometry data is used as the object ID because of its uniqueness. The size of the mapping table is N * 4 bytes. The table is built after the geometry data are loaded.

The geometry data maintain geometry coordinates of spatial objects in main memory for CPU. The length of a geometry is determined by the number of coordinates. The geometry data is used to draw spatial object on display screen. It is loaded from disk to main memory. The geometry file is created by converting and merging the original TIGER/Line shape files. The internal structure of it is optimized for the windows graphics.

B. Initial Loading Process

There are two kinds of processes to access spatial object in the proposed method; initial loading process and searching process. The initial loading process reads spatial objects and sends MBRs of read spatial objects to GPU, when the application starts. Fig. 3 shows the sequence diagram of initial loading process.

The controller is a module that controls the sequence of the overall application. The geometry is a module that reads spatial objects from the geometry file which is converted and merged from the original TIGER/Line shape files. The GPU is a module that plays a key role in searching. The GPU module is written in CUDA C. CUDA C extends C by making the programmer to define kernel functions as well as typical C functions. The kernel function is executed in parallel by different CUDA threads. Only in the searching process the kernel function is used. The functions in the initial loading process are identical to the typical C functions; CudaAllocObject() and CudaLoadMBR().

CUDA C provides C functions that execute on the host to allocate and transfer data between CPU (host) memory and GPU (device) memory, such as cudaMalloc(), cudaMemcpy() and cudaMemset(). The CudaAllocObject function uses cudaMalloc() and cudaMemset() functions to allocate the MBR table and the result flag set. The CudaLoadMBR function uses cudaMemcpy() function to transfer MBRs of spatial objects into GPU memory.

C. Searching Process

Once the memory structures are loaded, the application is ready to search for spatial objects with query region. The searching process is mainly executed on the GPU by calling CUDA kernel function. Fig. 4 shows the sequence diagram for the searching process.

The main controller module invoke CudaSearchObject() function with the query region in the form of MBR structure to execute CUDA kernel function. In the CudaSearchObject() function, the GPU module calculates the number of blocks *(B)* with the constant number of threads in a block *(T)* and the total count of spatial objects in the geometry data *(N)*, as in

$$B = \lfloor (N + T - 1) / T \rfloor. \qquad (1)$$

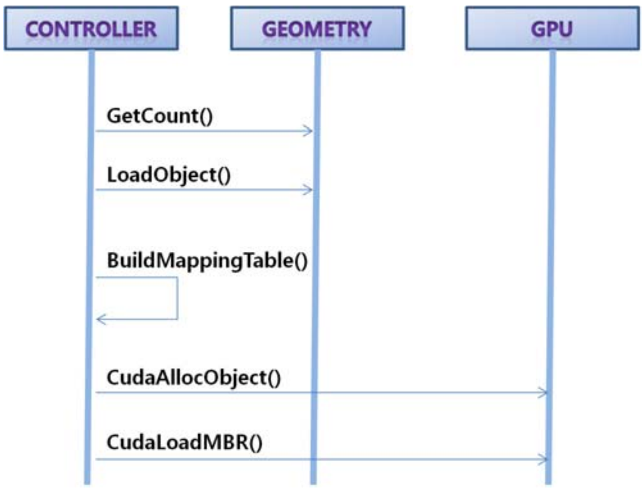


Figure 3. Sequence diagram for initial loading.

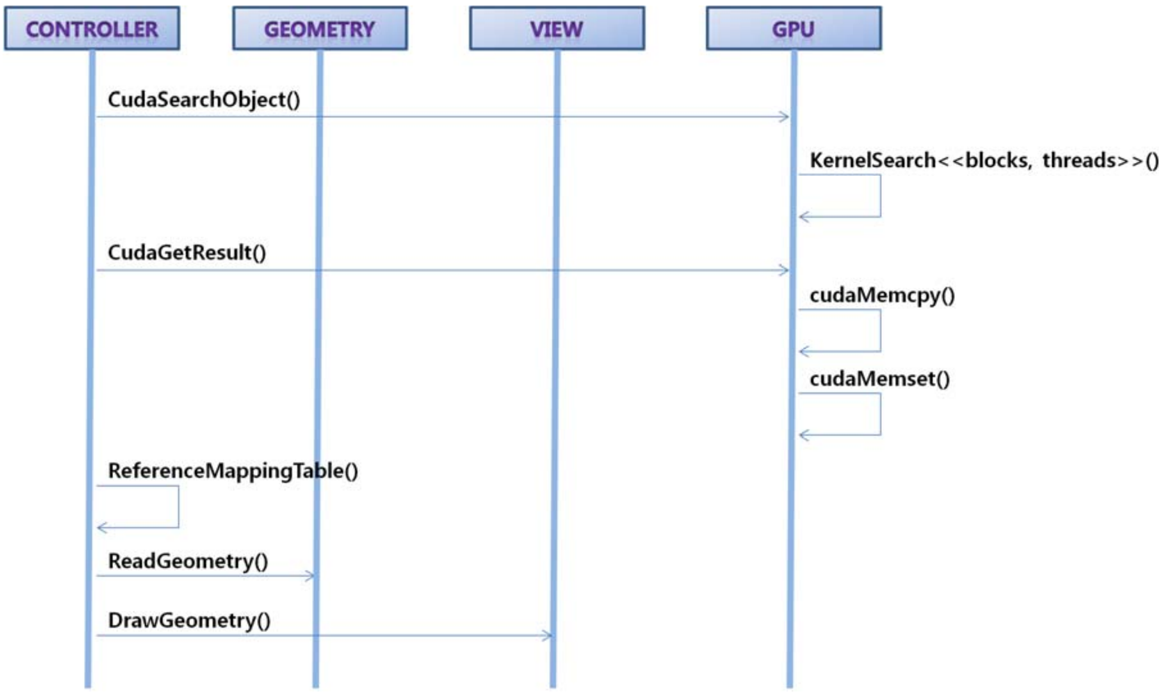
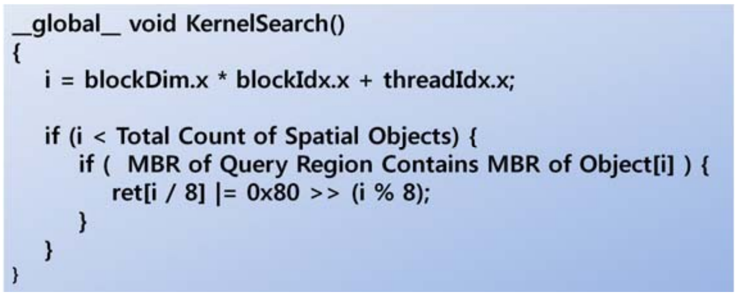


Figure 4. Sequence diagram for searching process.

The GPU module calls KernelSearch() function with execution configuration and function parameters. The execution configuration consists of two variables which are the number of blocks and the number of threads in the form of <<blocks, threads>>. The variable named "blocks" means the number of blocks *(B)* that is calculated in (1) and the constant variable "threads" means the number of threads in a block *(T)*. Because the number of threads in a block *(T)* is the fixed constant value, the last block may contain unnecessary threads. The KernelSearch() function has four parameters; the MBR of the given query region, the pointer of the MBR table array that is named as "Object", the pointer of the result flag set array that is named as "ret", and the total count of spatial objects which is *N* in (1). The KernelSearch() is implemented with a definition of CUDA kernel function as shown in Fig. 5.

The function starts with calculating the variable "i", that is an index to a spatial objects, with built-in variables which are explained in the chapter II, such as blockDim, blockIdx, and threadIdx. Because the last block may contain threads whose indexes are bigger than the total count of spatial objects, the function examines whether the value of calculated index "i" is less than the total count of the spatial objects or not. If a value of "i" of a thread is not less than the total count, the thread in the last block is invalid and should be ignored. If the MBR of an object is contained by the MBR of the query region, the thread that is mapped to the object sets the corresponding bit in the result flag set array to 1 with OR operation.

After the search, the controller module requests result of search by calling the CudaGetResult () function. The GPU module transfers the result flag set array from GPU memory to CPU main memory with cudaMemcpy() and re-initializes the result flag set in GPU memory for the next query with cudaMemset(). For each bit which is set to 1, the controller module references the mapping table to find the offset of the corresponding object in the geometry data and requests the spatial data to the geometry module by calling ReadGeometry() function. Finally, the view module draws the geometry of the found objects in the DrawGeometry() function.



```
__global__ void KernelSearch()
{
    i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < Total Count of Spatial Objects) {
        if (  MBR of Query Region Contains MBR of Object[i] ) {
            ret[i / 8] |= 0x80 >> (i % 8);
        }
    }
}
```

Figure 5. Pseudo-code of CUDA kernel function for searching process.

IV. EXPERIMENTAL RESULT

The experimental result were gathered from a machine running Microsoft Windows 7 with an Intel® Core™ i7 CPU running at 2.80GHz and 8 GB of memory. It was equipped with an NVIDIA GeForce GTX 580 with CUDA version 4.0, containing 1.5GB of memory and 512 cores with 16 multiprocessors.

A. Data Set

The data set for experiment is a set of the edges of California state whose layer type is all lines in the TIGER/Line data as shown in Fig. 6. The TIGER/Line files are a digital database of geographic features, such as roads, rivers, political boundaries, census statistical boundaries, etc. covering the entire United States. They was developed at the U.S. Census Bureau to support its mapping [12].

Fig. 6(a) shows the zoomed out picture of whole data set with scale 0.001 and (b) shows the zoomed in picture of a part of the data set which displays sample area in Sacramento, which is the capital city of the state of California, with scale 0.5. There are 4,077,020 spatial objects in the data set.

B. Query Region

In order to evaluate performance, query regions are randomly generated for various size as shown in Fig. 7. The shape of a query region is a square. The value in the Fig. 7 presents the size of the square. For example, Fig. 7(c) means that the length of one side of a query region is 1/50 of the length of the y size of the extent of the data set, i.e. 2% of the size Y. As the query region gets bigger, the number of found objects tends to be bigger. The number of the query regions is 1,000 for each size of query region.

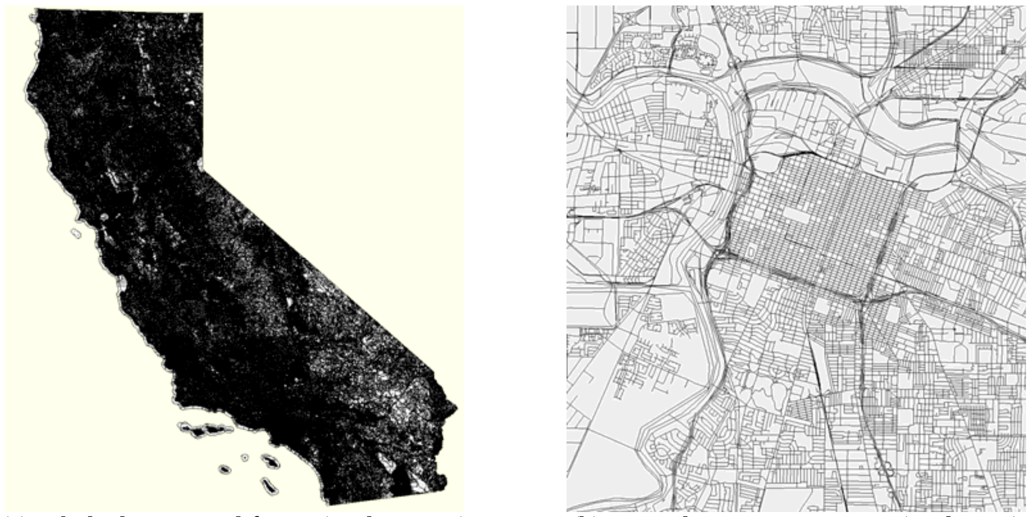
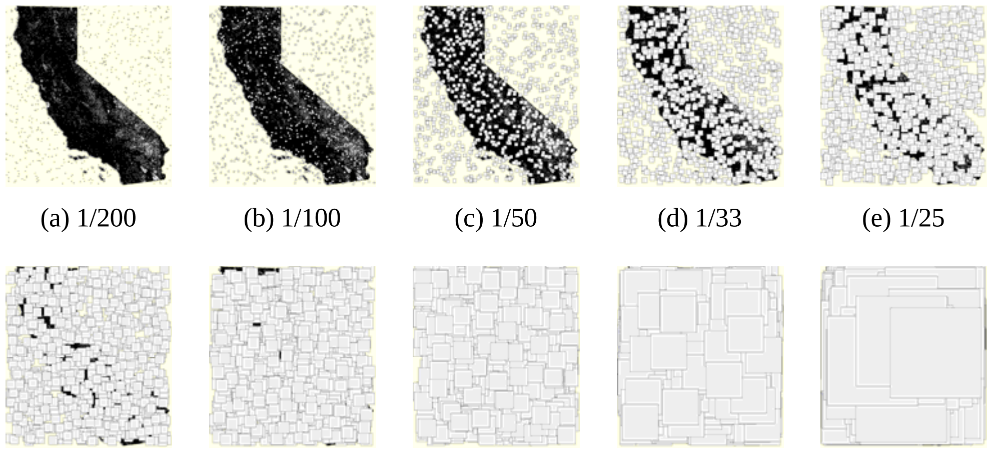

(a) Whole data set: California (Scale=0.001) (b) Example map: Sacramento (Scale=0.5)

Figure 6. Data set for the experiment.


(a) 1/200 (b) 1/100 (c) 1/50 (d) 1/33 (e) 1/25

(f) 1/20 (g) 1/15 (h) 1/10 (i) 1/5 (j) 1/2

Figure 7. Query regions for experiments with various size.

C. Result

To compare the performance of the proposed method, the main memory based R*-tree which is executed on CPU is compared. The queries were executed with the main memory based R*-tree on CPU, then with the proposed method on GPU, and the running times were compared. The execution time on GPU consists of search time and transfer time. The transfer time is required to transfer the result flag set from the GPU device to the CPU main memory, besides the search time. Table I shows the mean results for the 1,000 queries with various size from 1/200 to 1/2. The speed up is obtained by dividing the execution time for the R*-tree on CPU by the execution time for the proposed method on GPU with the same query size. The average count of found objects is the mean value of the number of result objects with the same query size.

The execution time of the main memory based R*-tree on CPU varied from 1.556ms to as high as 281.571ms with increasing of query size. The search speed of the R*-tree is still satisfactory considering 4 million objects. Only the problem is the wide range of variation in time. The execution time of the proposed parallel method in GPU has relatively narrow range of variation from 0.862ms to 1.510ms. The time to transfer the result data set from GPU to CPU main memory is around 0.3ms. The size of the result flag set is shrank to 498K bytes by using a bit for an object as mentioned earlier. Because the additional transfer time is an disadvantage in CUDA programming, this paper uses bit-wise operation to reduce the size of data to be transferred. The ranges of variation of two methods is depicted in Fig. 8.

As can be seen in Fig. 8, the proposed method using GPU is much faster than the memory based R*-tree approach executed on CPU. The R*-tree's search cost increases with the size of query regions, whereas the proposed parallel access method takes almost constant time. The reason for this behavior can be explained by the difference between parallel access and sequential access. The proposed access method is fully parallelized by using SIMT architecture of CUDA. In contrast, the R*-tree sequentially and iteratively executes node search at each tree level. Another reason for good performance is coalesced memory access. The proposed method's access pattern is completely coalesced, because of the sequence of the MBR table as mentioned in the chapter III. The experiment showed that the parallel method improves performance from 1.81 to 186.47 times faster compared to the R*-tree approach.

TABLE I. COMPARISON OF PERFORMANCE

Query Size	CPU (ms)	GPU (ms)			Speedup	Average Count of Found Objects
		Total	*Search*	*Transfer*		
1/200	**1.556**	**0.862**	0.581	0.281	**1.81**	99
1/100	**1.817**	**0.847**	0.563	0.284	**2.15**	465
1/50	**2.383**	**0.860**	0.568	0.293	**2.77**	2,350
1/33	**2.800**	**0.872**	0.570	0.302	**3.21**	6,237
1/25	**3.460**	**0.881**	0.577	0.304	**3.93**	12,917
1/20	**5.162**	**0.887**	0.587	0.300	**5.82**	26,230
1/15	**7.432**	**0.898**	0.596	0.303	**8.28**	49,889
1/10	**13.873**	**0.916**	0.613	0.303	**15.15**	104,609
1/5	**46.214**	**0.999**	0.692	0.307	**46.26**	326,072
1/2	**281.571**	**1.510**	1.188	0.321	**186.47**	1,867,896

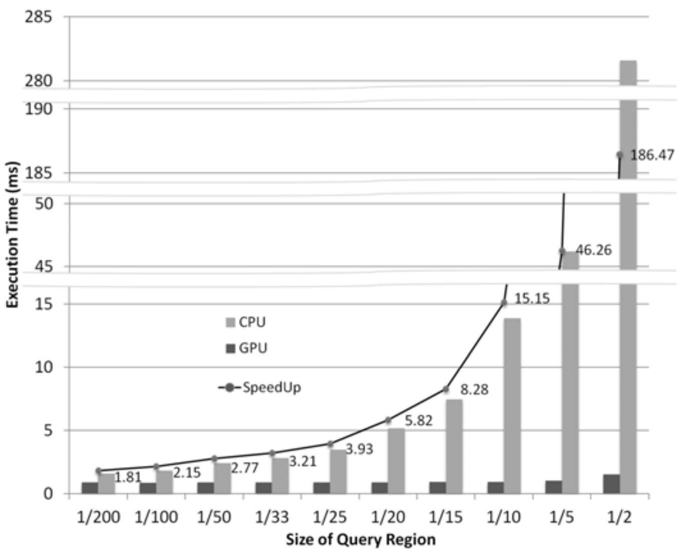


Figure 8. Query regions for experiments with various size.

V. CONCLUSION

This paper presented an parallel processing for fast finding of a large set of spatial data that are contained in a given query region using CUDA. The proposed method achieves high speed by efficiently utilizing the parallelism of the GPU throughout the whole process. The main work of this paper focused on the following aspects: utilizing the parallelism and reducing transfer time. Since the searching process needs intensive computation but is independently examined on a lot of MBRs of spatial data and CUDA provides SIMD with SIMT architecture, the spatial search function can be efficiently computed on GPU in a massive parallel way. The CUDA kernel function is implemented for utilizing the parallelism.

In general, the disadvantage of CUDA is that the additional transfer time is needed to copy data between CPU and GPU. This paper tried to reduce the disadvantage for maximum performance with two techniques: initial loading for the MBR table and bit-wise operation for the result flag set. Whenever the searching process is executed, the parallelized search thread needs the MBR table to examine the MBR of the spatial object. The initial loading process reads spatial objects and sends MBRs of read spatial objects to GPU, when the application starts. The loaded MBR table resides in the GPU memory until the end of application. Furthermore, the proposed method's access pattern for the MBR table is completely coalesced. It maximizes the performance. After the searching process, the result should be sent back to CPU to manipulate them. In order to reduce the transfer time, the result flag set is used to hold information about found objects. It is an array of bits that indicate whether the corresponding objects are selected or not.

The method achieves high speed by efficiently utilizing the parallelism of the GPU and by reducing transfer latency between GPU and CPU. In order to measure the speedup achieved by the proposed parallel method described in this paper, the execution time is compared with the sequential R*-tree that is loaded in the CPU main memory. With respect to the speedup, the proposed method achieved near linear for large queries and improved performance from 1.81 to 186.47 times faster than the main memory based R*-tree.

Future research could focus on the use of the cached GPU memory to draw the geometry data of the found object to reduce transfer latency between CPU and GPU when displaying the result after searching process.

ACKNOWLEDGMENT

This paper was supported by Research Fund (2009-104-061), Kumoh National Institute of Technology.

REFERENCES

[1] GPGPU Website, General-Purpose Computation on Graphics Hardware, http://gpgpu.org.

[2] NVIDIA, NVIDIA CUDA™ C Programming Guide (Version 4.0), 2011.

[3] P. Danilewski, S. Popov, and P. Slusallek, "Binned SAH Kd-Tree Construction on a GPU," Saarland University, June 2010, pp.1-15.

[4] E. Gaburov, J. Bedorf, and S.P. Zwart, "Gravitational Tree-code on Graphics Processing Units: Implementation in CUDA," Procedia Computer Science, Vol.1 Issue 1, ICCS 2010, pp.1119-1127.

[5] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," Journal of Computational Physics, Vol.228, Issue 12, 1 July 2009, pp.4468-4477.

[6] Murray, L., "GPU Acceleration of Runge-Kutta Integrators," IEEE Transactions on Parallel and Distributed Systems, Vol.23, Issue 1, 2012, pp.94-101.

[7] A. Krishnamurthy, S. McMains, K. Haller, K., "GPU-Accelerated Minimum Distance and Clearance Queries," IEEE Transactions on Visualization and Computer Graphics, Vol.17, Issue 6, 2011, pp.729-742.

[8] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, D. Manocha, "Memory-Scalable GPU Spatial Hierarchy Construction," IEEE Transactions on Visualization and Computer Graphics, Vol.17, Issue 4, 2011, pp.466-474.

[9] A. Guttman, “R-trees: A Dynamic Index Structure for Spatial Searching,” Proc. of ACM SIGMOD Intl. Conf., 1984, pp.47–57.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. of SIGMOD Intl. Conf., 1990, pp.322-331.

[11] AMD, AMD Accelerated Parallel Processing OpenCL Programming Guide (Version 1.3f), 2011.

[12] U.S. Census Bureau, TIGER products website, http://www.census.gov/geo/www/tiger.

AUTHOR PROFILE

Byoung-Woo Oh received his B.S., M.S. and Ph.D. degree in computer engineering from the Kon-Kuk University, Korea in 1993, 1995 and 1999 respectively. From 1999 to 2004, he was a senior researcher in the Electronics and Telecommunications Research Institute (ETRI), Korea. He is an Associate Professor in Department of Computer Engineering, Kumoh National Institute of Technology, Korea since 2004. His research interests include spatial database, automotive telematics, Geographic Information System (GIS) and Locatioin Based Service (LBS).