In-Memory Grid Files on Graphics Processors

Ke Yang, Bingsheng He, Rui Fang, Mian Lu, Naga Govindaraju*, Qiong Luo, Pedro Sander, Jiaoying Shi†

HKUST, China
{keyang, saven, rayfang, mianlu, luo, psander}
@cse.ust.hk

*Microsoft Corporation, USA
nagag@microsoft.com

†Zhejiang University
jyshi@cad.zju.edu.cn

ABSTRACT

Recently, graphics processing units, or GPUs, have become a viable alternative as commodity, parallel hardware for general-purpose computing, due to their massive data-parallelism, high memory bandwidth, and improved general-purpose programming interface. In this paper, we explore the use of GPU on the grid file, a traditional multidimensional access method. Considering the hardware characteristics of GPUs, we design a massively multi-threaded GPU-based grid file for static, memory-resident multidimensional point data. Moreover, we propose a hierarchical grid file variant to handle data skews efficiently. Our implementations on the NVIDIA G80 GTX graphics card are able to achieve two to eight times' higher performance than their CPU counterparts on a single PC.

1. INTRODUCTION

Multidimensional access methods, such as grid files [16] and R-trees [10], usually involve more complex data structures as well as more computation- and data-intensive operations than single-dimensional ones. For such multidimensional access methods, new generation graphics processors (GPUs) are a promising hardware platform due to their high memory bandwidth and massively parallel computation. For instance, in an NVIDIA G80 GTX graphics card, there are 16 multiprocessors, each containing 8 processors and supporting up to 512 threads. The observed overall performance is 330 GFLOPS and the device memory (of a size 768MB) bandwidth 86GB/sec.

Encouraged by the hardware features of GPUs, we study their use on the grid file, a representative multidimensional point access method. As a first step, we look at static multidimensional point data, such as those in the On-Line Analytical Processing (OLAP) environments or in CAD (Computer-Aided Design). These environments are query-intensive and have infrequent reorganizations on the data. Furthermore, we assume such data are brought into the GPU device memory from the main memory before access and are device memory resident throughout the query time.

Targeting at in-memory static data, we present a GPU acceleration of grid file for multidimensional point queries. To efficiently handle data skews, we adopt a hierarchical strategy by recursively constructing a sub-grid for a skewed cell that contains a large number of points. We have implemented the grid file on the CPU and the GPU. Our implementations achieve two to eight times speedup on the G80 GPU compared with their CPU counterparts.

This paper makes the following three contributions. First, we adapt the traditional, CPU-based grid file structure to fit for in-memory parallel environments, and provide a massively multithreaded GPU-based design. Second, we propose a hierarchical grid file variation that handles skewed data efficiently. This variation works both on the CPU and on the GPU, with a more significant performance improvement on the GPU due to GPU's inherent parallelism. Third, we empirically evaluate our GPU-based implementations in comparison with their CPU-based counterparts using an off-the-shelf PC equipped with the G80 graphics card.

The remainder of the paper is organized as follows. In Section 2, we briefly review the grid file structure, database processing on GPUs and the programming features of new generation GPUs. In Section 3, we describe the mapping of the basic grid structure on GPUs. In Section 4, we describe the hierarchical grid structure for skew handling. We present our experimental results in Section 5 and conclude in Section 6.

Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), June 15, 2007, Beijing, China.

2. BACKGROUND

Multidimensional Access Methods

Multidimensional access falls into two categories [6]: point access, which searches multidimensional points, and spatial access, which handles extended objects such as polyhedra. As a start to study the multidimensional access, we focus on point access methods. The following are three typical kinds of point access queries and their examples:

Exact match query. In such a query, the values of all attributes are given in equality predicates, and the query result is the record that exactly matches all the attribute values. E.g., find the student seated at Row 5, Column 3.

Partial match query. Such a query is a generalization of the exact match query. All predicates in a partial match query are equality predicates, but some attributes of the data points are absent in the query. Therefore, a partial match query retrieves all records that match on the specified attributes. E.g., find all students seated in Row 5.

Range query. A range query specifies a d-dimensional query box using range predicates and retrieves all records whose attributes represent a d-dimensional point located in the query box. E.g., find all students seated between Rows 1 to 3 and Columns 2 to 5.

There have been various multidimensional point access methods, including hashing-based methods [4][13][14][27], tree-structured methods [5][22][24] and space-filling curves [23]. Generally, the storage of hashing-based structures can be easily distributed, which is suitable for parallelization. Moreover, hashing-based methods require a constant access time to retrieve a record,

whereas tree structures often require a logarithmic function of the data size. Considering the GPU's parallel computation and high-bandwidth memory access features, we investigate further into hashing-based methods.

Grid Files

According to the classification by Geade et al. [6], the grid file [16] is a hashing-based multidimensional access method, even though the units of each dimension of a grid can be determined by range partitioning. There have been a number of variants of the grid file [3][12][18][25][28], and some parallelized methods [15][17]. In its basic form, a grid file superimposes a d-dimensional orthogonal grid on the d-dimensional data space. It partitions the space into hyper-rectangular cells using splitting hyper-planes that are parallel to the axes. These splitting hyper-planes follow the data distribution and the splitting positions may not be uniform across dimensions. As such, the positions along each axis indicating where to split are maintained in an ordered array called a *scale*. Finally, there is a *grid directory* to associate each grid cell to a bucket in the storage.

Figure 1 illustrates a 2D grid file. When accessing a point through the grid, we first use the scales to locate the cell that the point falls in, and then follow the pointer in the cell to the bucket in the storage, and scan the bucket for a match.

Traditional grid files are able to handle dynamic insertions and deletions. They split overflowed cells and merge underoccupied cells, to distribute the data to the buckets evenly so that a single record retrieval can be answered in at most two storage accesses. However, the splitting may lead to superlinear directory growth [21], and the merging can result in deadlocks [12][24]. Since we deal with static, memory-resident data in this paper, we leave dynamic insertions and deletions as future work.

Database Processing on GPUs

There has been intensive research on general-purpose computing using the GPU (GPGPU) [19]. One branch of our particular interest is on GPU-based database processing [2][7][8][9][26]. The existing work mainly utilizes the 3D graphics pipeline by drawing primitives such as a quad with OpenGL/DirectX programs. Our work, in contrast, exploits the most recent advance of the graphics hardware, and is implemented as general-purpose computing programs without utilizing any graphics APIs. Free from the constraints given by the graphics pipeline, our design and implementation is much more practical and flexible. Furthermore, to our best knowledge, this work is the first to develop multidimensional index structures on GPUs.

Programming on New Generation GPUs

We take the NVIDIA GeForce 8800 series (G80) graphics card, which was available on the market starting from Nov 2006 and has been used in our implementation, as an example to introduce the new generation GPUs. In comparison with its most advanced predecessors, G80 has made significant improvements for general-purpose processing. The computing resource consists of tens of SIMD multiprocessors, each of which contains a group of processors and registers that support a massive number of concurrent threads. Processors in the same multiprocessor share a cache called the *shared memory*, which is fully exposed to the programmer. The device memory could be accessed as textures in traditional graphics applications. Furthermore, it can be accessed as *global memory* in general-purpose computing programs, in a way similar to main memory. In addition, there is a *constant memory* that is shared by all multiprocessors and is cached on each multiprocessor.

The G80 card is released with a Compute Unified Device Architecture (CUDA) [1] for general data-parallel computing. CUDA provides a programming interface as an extension of the C language, with a runtime library for multithreaded parallel computing. This API treats the GPU as a general-purpose computing device as opposed to a programmable graphics pipeline, thus allows non-graphics researchers to utilize the GPU hardware features easily. Specifically, CUDA allows the programmer to specify the usage of the GPU resources such as the number of thread blocks (groups), and to write kernel programs that are executed on all threads.

3. BASIC STRUCTURES

In this section, we present the mapping of the grid file structure on the GPU and describe its construction and query processing.

3.1 Construction

To build a static grid file from a given data set, we first partition the data space in order to balance the bucket size of each cell as much as possible. Denote the numbers of splits along the d dimensions as $p_1, p_2, \ldots, p_d$, respectively, then the total number of grid cells $c = p_1 p_2 \ldots p_d$. Given the average number of records in each bucket, H, we have $cH = N$, where N is the number of records of relation R.

We build a grid for R and rearrange R such that the records belonging to the same cell are clustered into one bucket. First, we obtain a scale in each dimension i $(i=1, 2, \ldots, d)$ by sorting R along that dimension and sampling p_i quantiles as the elements of the scale. Then, for each record, we use the scales to identify the bucket it belongs to. This procedure is done in the LocateCell routine, which performs binary search into each ordered scale array for the location of the record. Second, to get the starting position of each bucket in R, we build a histogram of number of records in each bucket. This is done by scanning R once and using the scales to identify the bucket each record belongs to. Third, a prefix sum routine translates the histogram into bucket offsets in the rearranged relation. Then the records are scanned again to be scattered into the corresponding buckets with given starting offsets. After the construction, the rearranged R stores the buckets, the grid directory entries contain the bucket offsets, as illustrated in Figure 1. In this example, $d = 2$, $N = 10$, $H = 2$, and $p_1 = p_2 = 2$.

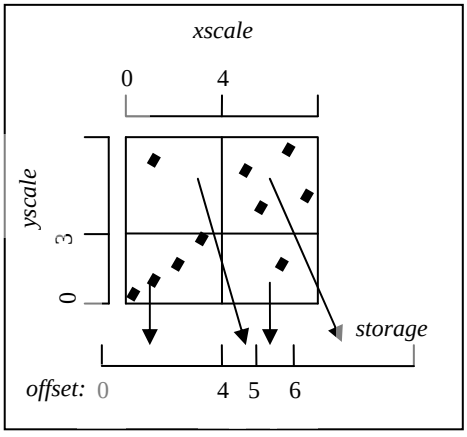

Figure 1. Structure of a 2D grid file.

We have implemented the construction with the GPU co-processing on the sorting step. On a relation consisting of 16 million 2D records, the pure CPU-based construction takes about 12 seconds, among which 8 seconds are spent on sorting. We employ the GPU sort primitive [11] and reduce the sorting to 3 seconds.

3.2 Query Processing

After a grid file is ready on the GPU, we can process the three kinds of multidimensional point queries using the grid, namely the exact match queries, the partial match queries, and the range queries. Note that a partial match query is similar to an exact match query on the equality predicates and to a range query on an unbounded attribute where a range box extends throughout the entire domain of the attribute. As a result, our implementation on the processing of partial match queries is similar to that of the other two types. In the following, we will mainly discuss the exact match and the range queries.

Since the GPU is a data-parallel computing device, we use a pool of threads, which is tuned to fully utilize the hardware computation power, to handle a large number of queries in parallel. Each thread takes charge of a query independently, and after finishing one, it handles another. In specific, the thread with index t starts its i-th query by reading in the $(t + iT)$'s query from the device memory, where T is the number of threads. This strategy of assignment is called *coalesced read* [1], which can speed up the device memory access.

For an exact match query, a thread scans the bucket corresponding to the cell that contains the query record to search for a match. This search is achieved by a LocateCell followed by a sequential scan in the bucket. The termination condition of the scan, i.e., the boundary of the bucket, is marked by the offset of the next bucket.

For a range query, a thread scans all the buckets that correspond to the cells that overlap the query box. Given the two end points, L and H, of the major diagonal of the box, the thread calls LocateCell on L and H to obtain the two corresponding end cells, CL and CH. The coordinates of CL and CH bound the cells whose coordinates fall in the multidimensional range. Then the thread sequentially scans these cells. For the points in the boundary cells (those having at least one coordinate equal to that of an end cell), the thread further takes a point-level test to check if they are located in the query box.

To avoid the conflict among multiple threads that concurrently write query results to the shared output region, we perform the write in a three-step scheme similar to that in our previous work [11]. For completeness, we briefly present the scheme here. In the first step, each thread executes the query and counts the number of query results it generated. In this step, each thread only outputs the count but does not produce the actual query result records. Then a prefix sum routine gathers these local counts and translates them into an array of global write locations, each of which contains the start position for the corresponding thread to output. In the last step, each thread computes the query result records and writes the results to its slot in the global memory.

4. SKEW HANDLING

4.1 Hierarchical Grid File

When partitioning the data space in the construction process of a grid file (described in Section 3.1), the data distribution in the resulting buckets may not be necessarily balanced after a single partitioning pass. For example, the two buckets starting from 0 and 6 in Figure 1 are more crowded than the other two. As a result of data skews, querying a grid cell that corresponds to a crowded bucket is more expensive than querying a less crowded one. This imbalance has significant performance implications on the GPU, because the processors of the GPU are SIMD. Therefore, we propose a hierarchical scheme to further divide the crowded cells. This division is done recursively until the bucket size of a resulting cell is below a given threshold.

Our proposed hierarchical grid structure is similar to two existing schemes, the multilevel grid file [28] and the buddy tree [24]. The common idea among the three schemes is to divide crowded regions recursively. However, our scheme has two major differences from the existing work. First, both existing structures cover only those cells that contain data points, and maintain a directory entry for each non-empty cell. In contrast, our hierarchical grid covers the entire data space, and locates cells through shared scales. This structure is relatively simpler and more suitable for bulk loading in a parallel computing environment. Second, as dynamic maintenance techniques, the two existing methods split an overflowed bucket into two at each level, thus the structures contain a relative large number of levels in the tree or in the grid. In comparison, our hierarchical grid is a static structure, and the number of levels of sub-grids in a crowded cell is relatively small.

To build a hierarchical grid for relation R, we first perform the splitting and the data rearrangement of R in the same way as described in Section 3.1. We then store the information about the grid, such as the scales, the bucket sizes and offsets, in the directory. Next we check the size of each bucket in this grid: if the size of a bucket is larger than a given threshold, we perform another round of splitting, and the information of the newborn sub-grid is appended to the directory. The offset of the parent bucket is now redirected to the offset of the sub-grid in the directory. To distinguish the offset of a sub-grid from that of a bucket, we add a sign bit flag to the offset of the sub-grid. The pseudo code for constructing a hierarchical grid is given in Figure 2. The average bucket size H is pre-specified as a constant. For simplicity, here the routine PNum assumes the same number of splits on all dimensions, i.e. $p_1 = p_2 = \ldots = p_d = p$. This can be generalized to cases where $p_1, p_2 \ldots p_d$ are functions of p.

```
int PNum(int size) // decide the number of partition on a dimension
{
    p = 2;  // number of partitions
    while(size/H > p^d) p++; // p^d: total number of cells
    return p;
}
BuildGrid(void* dir, rec R[n], int p)  // build a grid directory for R
{
  compute scales, bucket offsets and sizes; append them to dir;
  rearrange R;
  for each bucket i of size[i]/H > 2^d // the threshold to split
        BuildGrid(dir, R + offset[i], PNum(size[i]));
        offset[i] = 0 - dir's current offset;
}
```

Figure 2. Pseudo code for building a hierarchical grid file.

An example hierarchical grid is illustrated in Figure 3, obtained through splitting the original grid in Figure 1. After identifying the two crowed cells, we construct a sub-grid for each cell and change the offset in the storage into the flagged offset in the directory, i.e., -12 and -24.

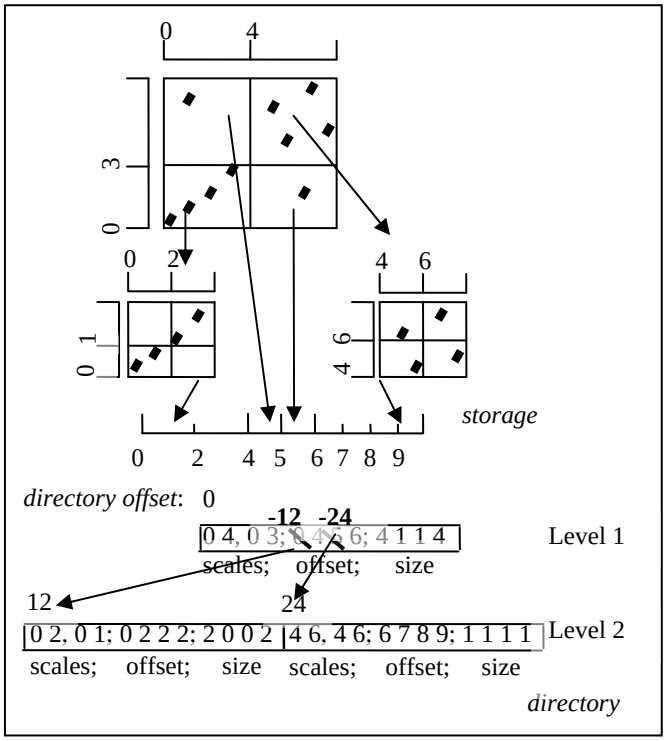

Figure 3. Structure of a hierarchical grid file.

4.2 Query Processing

Query processing over a hierarchical grid is similar to that over an original grid, except that each thread recursively decodes the offset of a sub-grid in the directory until it reaches the final bucket. Pseudo code is given in Figure 4. Note that, recursion is not supported in a GPU kernel program, due to the hardware limitations. In our implementation, we rewrite the code to a *while* loop with offset[i]<0 as the condition. Furthermore, flow control instructions can cause threads to diverge, and different execution paths have to be serialized. Since our grid hierarchy usually has a low level, the kernel requires only a small number of branches (less than five in our tests). Additionally, the hierarchy greatly improves the worst cases of bucket imbalance, thus effectively limits the serialization cost.

```
Search(rec q, int cur) //search q from current position of directory
{
    i = LocateCell(dir + cur, q);  // bucket id
    if(offset[i] > 0)  // a bucket
        scan the bucket for a match;
    else  // a sub-grid
        Search(q, - offset[i]); // grid offset
}
```

Figure 4. Pseudo code for query in a hierarchical grid file.

5. EXPERIMENTS

5.1 Experimental setup

We have implemented and tested our algorithms on a PC with a G80 GPU and an Intel P4 Dual-Core processor running Windows XP. The hardware configuration of the PC is shown in Table 1.

Table 1. Hardware configuration

	GPU	CPU
Processors	1350MHz × 8 × 16	3.2GHz (Dual-core)
DRAM (MB)	768	1024
DRAM latency (cycle)	200-300	300
Bus width (bit)	384	64
Memory clock (GHz)	1.8	0.8

Each multiprocessor on the GPU has a piece of constant memory sized 64KB. The accesses to the constant memory are cached. The constant cache on each multiprocessor is 8KB. Since the scales of the grid file are frequently accessed, we store the scales of the first level of the grid in the constant memory for fast access. In our GPU programs, the numbers are tuned for the best performance. We use a configuration of 5120 thread blocks, each containing 256 threads.

We consider two kinds of workloads in our experiments, the exact match query and the range query. For exact match queries, we first generate uniform datasets with number of tuples and number of dimensions varied. Then we test on skewed data sets, some synthetic, and some real-world. For range queries, we use a uniform dataset and vary the selectivity of the range query.

The synthetic skewed data follows the *Gaussian* distribution with the parameter *standard deviation* varied. The smaller the standard deviation, the more skewed the data distribution is. The real-world skewed data sets are from 3D point cloud models, which have been used extensively in graphics and computational geometry studies.

The data structure for a *d*-dimensional record is an integer id followed by *d* 32-bit unsigned integer keys. We set the average bucket size to be *H=8* in all experiments. In each experiment, the time cost for the CPU and the GPU executions are separately measured for comparison.

5.2 Results

5.2.1 Exact match query

Figure 5 demonstrates the performance of evaluating 1 million exact match queries on a relation with the number of tuples varied. As the number of tuples increases, the performance speedup of the GPU-based algorithm over the CPU-based one increases slightly. In particular, the performance speedup is 6.5x on the data set of one million tuples and is 7.7x on the one of 16 million tuples. We also test the GPU grid file without the optimization of storing scales into the constant memory, and the comparison shows this optimization greatly reduces the memory stalls, and improves the overall performance by 40% on average.

Figure 6 shows the performance of evaluating 1 million exact match queries on a relation of 16 million tuples with the number of dimensions varied. Because the overhead of LocateCell is proportional to the number of scale arrays, the time cost increases with the number of dimensions. The GPU-based grid file is 2-5 times faster than the CPU-based grid file.

Figure 7 shows the measurements on the synthetic skewed data, with the standard deviation ranging from 10^3 to 10^7. Both CPU- and GPU-based implementations suffer when the data skew is severe. As the data is becoming less skewed, the maximum level of the hierarchical grid decreases accordingly. The GPU-based hierarchical grid file is generally more than five times faster than the CPU-based one.

Finally, we evaluate the exact match query on the skewed data using 3D point cloud models. We use two models, *Sphere* and *Dragon*, as shown in Figure 8. We varied the number of points in each model from 1 to 16 million, and issued 1 million queries with random search keys. The performance results on these two models are shown in Figure 9 and Figure 10, respectively. On both the CPU and the GPU, we compare the performance with and without the hierarchical scheme, denoted as "Y" or "N" for with or without a hierarchy, respectively. In general, the GPU versions are 2x-5x faster than their CPU counterparts.

For the sphere model (Figure 9), the performance of the grid file with the hierarchical scheme is similar to that without, both on the CPU and on the GPU. This performance similarity is because the sphere model is a uniformly distributed point cloud within a sphere, which is of low skewness. The maximum level of the grid file is one. This shows that our partition scheme can handle slightly skewed data in a single-level grid. Similar to the performance speedup of GPU over CPU on the uniform data set, the performance speedup on the sphere model increases slightly as the number of points in the model increases.

The dragon model is the point cloud on the surface of a dragon, which is of high correlation and skewness. The maximum level in this grid file is 3. The CPU-based grid file has an improvement of 1.2x-1.5x by utilizing a hierarchy, whereas the GPU-based grid file gains an improvement of 2.3x-4.5x by utilizing a hierarchy. The main reason for this different degree of improvement is that the GPU benefits more from load balance than the CPU does. On the GPU, when threads are severely load-unbalanced and thus greatly diverged, the less loaded threads have to wait for the busy threads. Since the hierarchy helps balancing the load, the waste on waiting among threads is largely reduced. Thus the GPU benefits much from load balance.

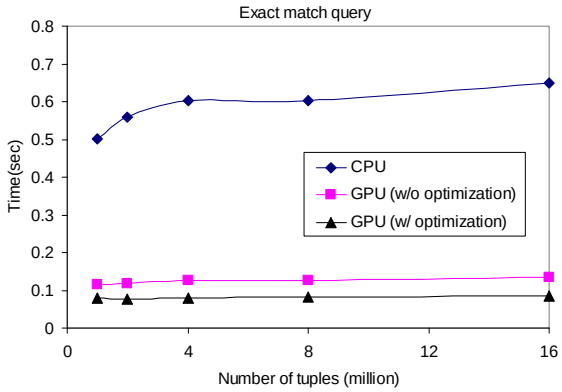

Figure 5. Exact match query on uniform data sets with the number of tuples varied.

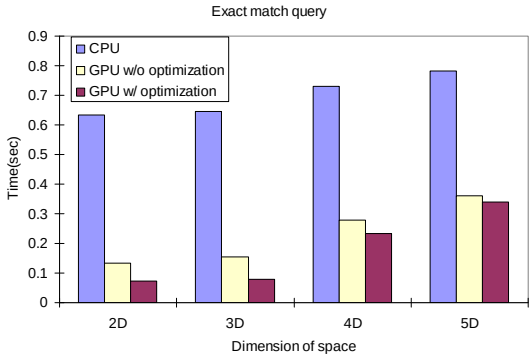

Figure 6. Exact match query on uniform data sets with the number of dimensions varied.

5.2.2 Range query

Figure 11 shows the performance of evaluating 100k range queries to the grid of a relation of 16 million randomly generated 2D tuples. Both the width and the length of the rectangles are varied from 0.1% to 1% of the integer range. As the selectivity increases, the execution time of both the CPU- and GPU-based grid files increases almost linearly. The GPU-based grid file is around 4x-6x faster than the CPU-based one.

5.2.3 Discussion

We have shown that our GPU-based algorithm outperforms its CPU counterpart in all our tests, with a 2x-8x speedup. The reasons for the performance improvement are as follows: (1) We utilize the GPU as a parallel device with a large number (more than 1 million) of lightweight threads. This massive-threading model well matches the query-intensive workloads. (2) Our GPU-based grid file structure is relatively simple, and the record type is regular. This storage structure fits the array-based GPU memory access. (3) Each single query operation is relatively simple, and the hierarchical structure further improves the load balance. Such workload takes full advantage of the SIMD GPU processing and alleviates the high cost on branches or inter-thread load unbalancing. For these reasons, the GPU is more suitable for grid files than a multi-core CPU, which is equipped with a powerful instruction set but executes a small number of heavyweight threads.

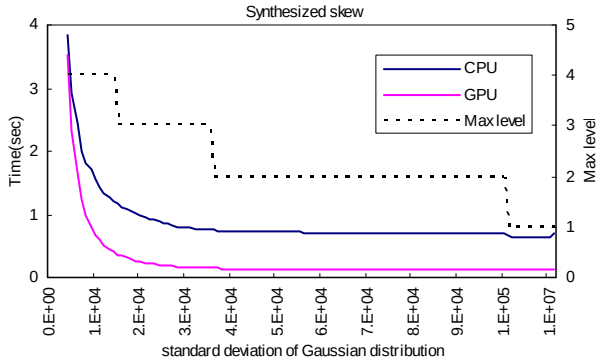

Figure 7. Exact match query on skewed data sets with the *standard deviation* in the Gaussian distribution varied.

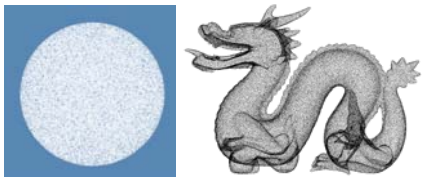

Figure 8. Visualization of the 3D real-world datasets: (left) Sphere; (right) Dragon.

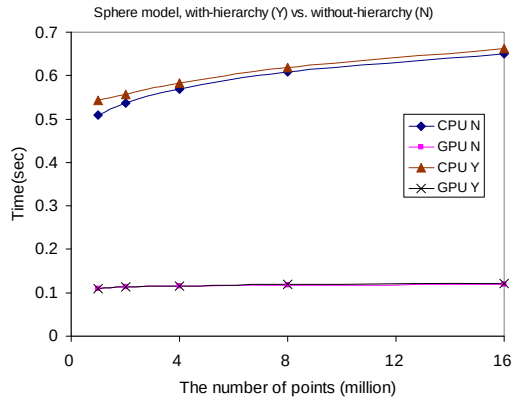

Figure 9. Exact match query on the Sphere model with the number of points varied.

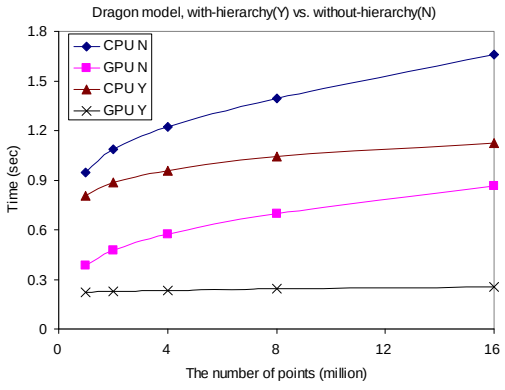

Figure 10. Exact match query on the Dragon model with the number of points varied.

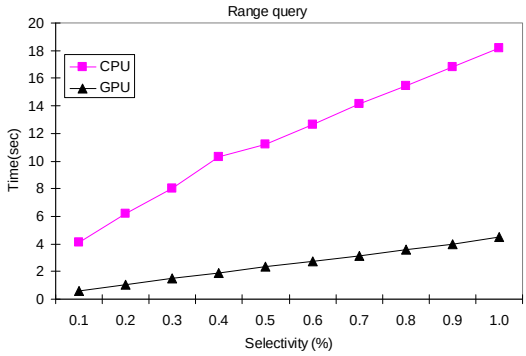

Figure 11. Range query on uniform data set with the selectivity varied.

6. CONCLUSIONS AND FUTURE WORK

We have developed in-memory grid files on the GPU and have shown that the new generation of GPUs is a well-suited parallel platform for accelerating this traditional multidimensional point access method. Moreover, we have proposed a static hierarchical grid file structure that handles skewed data efficiently. Experimental results show that our GPU algorithms greatly outperform their CPU counterparts in processing exact match and range queries, and that they work well up to five dimensions. As future work, we plan to study alternatives for dynamic insertion and deletion operations for grid files. Also, we are interested in designing multidimensional spatial access methods, such as R-trees [10], on GPUs.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank people at the NVIDIA CUDA Forum, especially Mark Harris, for their help with the G80 implementation issues. Finally, we thank Dr. Lidan Shou of Zhejiang University for his lectures on multidimensional access methods.

8. REFERENCES

[1] NVIDIA CUDA (Compute Unified Device Architecture), http://developer.nvidia.com/object/cuda.html.

[2] Bandi, N., Sun, C., Agrawal, D. and El Abbadi, A., Hardware acceleration in commercial databases: A case study of spatial operations. VLDB, 2004.

[3] Blanken, H., Ijbema, A., Meek, P. and van den Akker, B., The generalized grid file: Description and performance aspects. In Proc. 6th IEEE Int. Conf. on Data Eng., pp. 380-388. 1990.

[4] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, R., Extendible hashing: A fast access method for dynamic files. ACM Trans. Database Systems 4 (3), 315-344. 1979.

[5] Finkel, R. and Bentley, J. L., Quad trees: A data structure for retrieval of composite keys. Acta Informatica 4(1), 1-9. 1974.

[6] Gaede V, Gunther O, Multidimensional Access Methods. ACM Computing Surveys,1998, 30(2).

[7] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D., GPUTeraSort: high performance graphics coprocessor sorting for large database management. SIGMOD, 2006.

[8] Govindaraju, N., Lloyd, B., Wang, W., Lin, M. and Manocha, D., Fast computation of database operations using graphics processors. SIGMOD, 2004.

[9] Govindaraju, N., Raghuvanshi, N. and Manocha, D., Fast and approximate stream mining of quantiles and frequencies using graphics processors. SIGMOD, 2005.

[10] Guttman, A. R-trees: A dynamic index structure for spatial searching. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 47-54. 1984.

[11] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q. and Sander, P., Relational Joins on Graphics Processors. Technical report, Department of Computer Science and Engineering, HKUST, March 2007.

[12] Hinrichs, K., Implementation of the grid file: Design concepts and experience. BIT 25, 569-592. 1985.

[13] Kriegel, H.-P. and Seeger. B., Multidimensional order preserving linear hashing with partial expansions. In Proc. Int. Conf. on Database Theory, Number 243 in LNCS, Berlin/Heidelberg/New York. Springer-Verlag. 1986.

[14] Kriegel, H.-P. and Seeger, B., Multidimensional quantile hashing is very efficient for non-uniform record distributions. In Proc. 3rd IEEE Int. Conf. on Data Eng., pp. 10-17. 1987.

[15] Li, J., Rotem, D., Srivastava, J., Algorithms for Loading Parallel Grid Files. SIGMOD Conference 1993: 347-356

[16] Nievergelt, J., Hinterberger, H., and Sevcik, K. C., The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Systems 9 (1), 38-71, 1984.

[17] Mohammed, S., Srinivasan, B., Bozyigit, M., Phu, D,. Novel parallel join algorithms for grid files. 3rd International Conference on High Performance Computing. Dec 1996, pp 144-149.

[18] Ouksel, M., The interpolation based grid file. In Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, pp. 20-27. 1985.

[19] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., A. E. Lefohn and T. J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, Volume 26, 2007.

[20] Rao, J. and Ross. K. A., Cache conscious indexing for decision-support in main memory. VLDB, 1999.

[21] Regnier, M. Analysis of the grid file algorithms. BIT 25, 335-357. 1985.

[22] Robinson, J. T. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 10-18. 1981.

[23] Sagan, H., Space-Filling Curves. Berlin/Heidelberg/New York: Springer-Verlag, 1994.

[24] Seeger, B. and Kriegel, H.-P., The buddy-tree: An efficient and robust access method for spatial data base systems. In Proc 16th Int. Conf. on Very Large Data Bases, pp. 590-601. 1990.

[25] Six, H. and Widmayer, P., Spatial searching in geometric databases. In Proc.4th IEEE Int. Conf. on Data Eng., pp. 496-503. 1988.

[26] Sun, C., Agrawal, D. and El Abbadi, A., Hardware acceleration for spatial selections and joins. SIGMOD, 2003.

[27] Tamminen, M. The extendible cell method for closest point problems. BIT 22, 27-41. 1982.

[28] Whang, K.-Y. and Krishnamurthy, R., Multilevel grid files. Yorktown Heights, NY: IBM Research Laboratory. 1985.