Indexing Large-Scale Raster Geospatial Data Using Massively Parallel GPGPU Computing

Jianting Zhang
Department of Computer Science
City College of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

Simin You
Department of Computer Science
City University of New York Graduate Center
365 Fifth Avenue, New York, NY, 10006
syou@gc.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT

Advances in geospatial technologies have generated large amounts of raster geospatial data. Massively parallel General Purpose Graphics Processing Unit (GPGPU) computing technologies have provided personal computers with tremendous computing capabilities. In this paper, we report our work on fast indexing of large-scale raster geospatial data using GPGPU computing. We have designed a cache conscious, pointerless quadtree data structure (CCQ-Tree) that has low memory footprint, is suitable for GPU indexing and allows fast mapping between main memory and hard drives. A set of algorithms have been developed and integrated to construct CCQ-Trees on GPU devices by utilizing multiple pyramid data structures and Z-order based prefix sum. Experiments on multiple 4096*4096 blocks of a global precipitation raster data have shown that CCQ-Tree indexing using an Nvidia Quadro FX3700 GPU device reduces construction times from around 9.83 seconds to 0.42 seconds (23X speedup). The experiment results have led us to project the possibility of real time indexing of global 30 arc-seconds (approximately 1km) resolution data (43200*21600) in 1-5 seconds on a personal workstation with 1-4 Nvidia next generation Fermi GPU devices.

1. INTRODUCTION

Advances in geospatial technologies, especially space-borne satellite remote sensing and large-scale environmental modeling, have generated large amounts of raster geospatial data at the ever increasing speeds. Different from vector geospatial data that sophisticated indexing and query processing techniques have been extensively developed and applied, research on indexing raster geospatial data is limited. We refer to the overview article by [18] and the comprehensive book by Samet [54] for more details. Traditionally raster geospatial data are mostly used for sophisticated offline analysis (such as image classification and physics based environmental modeling) and simple online display (such as tiling based image display in Google Map and Google Earth). Massively parallel General Purpose Graphics Processing Unit (GPGPU) technologies [21], which allows using graphics processing units for general purpose computing, have provided personal computers with tremendous computing capabilities [46][45]. For example, a single Nvidia Fermi-based GeoForce GTX480 GPU device has 480 cores and a peek floating point performance of 1.35 Terra Flops [67] at the cost of a few hundreds of dollars. The computing power at this level was only available on supercomputers a few years ago. It has been projected that, while CPU speed improvement will only be around 20% per year due to the combined power, memory and instruction-level parallelism problem [6], GPU speed improvement will be more than 50% per year, over the next decade [17]. The computing power provided by GPGPU devices will enable many traditionally offline applications to run online and interact with users.

ACM GIS 2010 San Jose, Calfornina, USA

In this study, we target at the two specific types of queries, i.e., spatial range queries and Region-of-Interests (ROI) type of queries, for large-scale raster geospatial data. A spatial range query (or window query) returns all spatial objects (including quadrants) that fall within a spatial query window. The properties or the values of the spatial objects can be retrieved based on the object identifiers. The ROI-type query returns all spatial objects (including quadrants) that satisfy one or more value range criteria, e.g., temperature between [t1,t2) and precipitation between [p1,p2). It can be seen that a spatial range query maps spatial objects to values while a ROI-type query maps values to spatial objects and they are complement to each other. Cascading these two types of queries on multiple rasters allows users to identify interesting patterns, such as, patterns related to the spatial distributions of geospatial phenomena with certain value thresholds (e.g., storms with precipitation greater than 10mm) and potential casual relations between multiple rasters through co-location analysis (e.g., biodiversity decrease and deforestation/climate changes across the globe).

At a first glance it seems that it is straightforward to utilize the massively parallel computing power provided by GPGPU devices for raster geospatial data due to the regular layouts of raster data. Indeed, many GPGPU based image processing techniques [57] and environmental models [36] have been developed since the debut of GPGPU computing technologies by taking advantage of per-pixel based algorithms, which can be effortlessly parallelized, in a way similar to computer graphics applications where GPU technologies originate from, such as iso-surface generation [63] and ray-tracing [29]. However, from a raster data processing perspective, most of existing techniques focus on local and focal operations, i.e., operations on a single raster cell or neighborhood of a single cell [60], which are relatively easy to parallelize. On the other hand, the irregularity of data layout in global and zonal operations often makes parallelization non-trivial and very little research on parallelizing zonal or global raster operations has been reported. We consider indexing raster geospatial data as a special global operation. More specifically we focus on quadtree based indexing due to its well-known data compression and pruning power in query processing [54]. Low footprint memory-resident quadtrees with nodes associated with proper derived information can be used to answer window queries and ROI-type queries efficiently by minimizing costly accesses to disks. We note that while it is feasible to perform window query by scanning relevant raster cells that fall within the query window in disk files and then retrieve desired information, it is advantageous to retrieve the desired information from main-memory resident tree indices directly without accessing data files on disks. This is especially beneficial to queries with large query windows where a large portion of data files need to be scanned without indices. As for processing ROI-type queries, it is inevitable to scan all the raster cells if no indices are available. For both window queries and ROI-type queries, with quadtree indexing, the query results will be returned as quadrants rather than individual raster cells which is desirable in many applications. Previous work has shown that assembling individual raster cells into regions could be very expensive [55] and may not be suitable for online interactive applications.

Different from local or focal raster operations that transform one raster to another, quadtree based raster data indexing transforms a regularly shaped grid into an irregular, hierarchical data structure. While the irregularity can be relatively easily handled on CPUs through dynamic memory allocations and pointer linking, as the current GPGPU computing does not support dynamic memory allocations and recursions, it is technically challenging to generate, store and manipulate tree data structures on GPUs. In this study, we propose a Cache Conscious Quadtree (CCQ-Tree) data structure for GPUs. CCQ-Tree completely eliminates pointers that are used in traditional main-memory based quadtrees by laying out the tree nodes as array elements. A CCQ-Tree can be easily mapped between main memory and disk without the expensive index tree reconstruction as required by traditional linear quadtrees [56]. We design a set of algorithms to generate CCQ-Trees on GPUs by utilizing a few pyramid data structures. Compared to constructing pointer-based main-memory quadtrees on CPU, GPU based CCQ-Tree construction has shown significant speedups. Our technical contributions can be summarized as follows:

- We design a cache conscious, pointerless quadtree data structure (CCQ-Tree) that has low memory footprint, is suitable for GPU indexing and allows fast mapping between main memory and hard disks.
- We develop a set of algorithms to construct CCQ-Trees on GPU devices based on GPGPU computing technologies.
- We perform experiments that show that CCQ-Tree indexing using GPGPU computing speeds up its construction more than 20 times on average using an Nvidia Quadro FX3700 GPU card. Higher speedups are expected using the next generation Fermi GPU devices when they become available.

The remainder of the paper is structured as follows. Section 2 introduces related works. Section 3 presents the CCQ data structure and its construction algorithm on GPUs. Section 4 presents our experiments on the WorldClim global precipitation data at the 30 arc-seconds (approximately 1km) resolution. Finally, Section 5 concludes the paper and outlines future research directions.

2. BACKROUND AND RELATED WORK

2.1 GPGPU-based Parallel Computing Architecture

A Graphics Processing Unit (GPU) is a hardware device that is originally designed to work with CPU to accelerate rendering of 3D or 2D graphics. The highly parallel structures of modern GPU devices, such as AMD/ATI Radeon [7] and Nvidia GeForce [66] series, make them more effective than general-purpose CPUs for a range of complex graphics-related algorithms. The concept of General Purpose GPU (GPGPU) turns the massive floating-point computational power of a modern graphics accelerator's graphics-specific pipeline into general-purpose computing power [21]. GPGPU computing technologies provide a cost effective alternative to cluster computing and have gained considerable interests in many scientific research areas in the past few years[46][45]. According to the Nvidia website, when compared to the latest quad-core CPU, Tesla 20-series GPU computing processors deliver equivalent performance at 1/20th of power consumption and 1/10th of cost [42]. As many reasonably current desktop computers have already equipped with GPGPU enabled graphics cards, GPGPU based processing of raster geospatial data can improve system performance significantly without additional costs. In this study, we utilize an Nvidia Quadro FX 3700 GPU card with 512M device memory that comes with a Dell T5400 workstation. Obviously a more powerful Tesla GPU card will speed up the performance even more but requires additional monetary cost. Despite the differences among the GPGPU enabled devices and development platforms, a GPGPU device can be viewed as a parallel Single Instruction Multiple Data (SIMD) machine with a limited instruction set [40]. Although our experiments are performed on an Nvidia GPU device and based on its Compute Unified Device Architecture (CUDA, [39]), we argue that our CCQ-Tree index construction algorithm can be easily adapted to other types of GPUs such as AMD/ATI Stream programming enabled ones [3]. We next briefly introduce the Nvidia GPU architecture and its parallel programming abstraction based on CUDA.

While different models of Nvidia GPU cards have different architectures, CUDA-enabled GPU devices are organized into a set of Stream Multiprocessors (SMs). Each SM has a certain number (e.g., 16 or 32) of computing cores. All the cores in a SM share a certain amount (e.g., 16k or 64k) of fast memory called shared memory and all the SMs have access to a large pool of global memory (e.g., 512M or 4G) on the device. According to CUDA, developers write a special C-like code segments called kernels. The kernels are invoked by the companioning CPU code to run on GPU devices. The kernel code does not allow dynamic memory allocation and recursion which imposes significant technical challenges for many database applications that rely on these techniques, including tree indices constructions. CUDA based GPGPU programming makes it easier for task and data decomposition and subsequent parallel computing. Basically a developer specifies the sizes of the layout of the data to be processed in the units of data blocks and the number of threads to be launched inside a data block. The GPU device is responsible for mapping the data blocks to the SMs through space and time multiplexing which is transparent to developers/users. Since each SM has limited hardware resources, such as the number of registers, shared memory and thread scheduling slots, a SM can accommodate only a certain number of blocks subjected to the combination of the constraints. Carefully selecting block sizes allows a SM to accommodate more blocks simultaneously and, subsequently, improve parallel throughputs.

2.2 Database Applications of GPGPU Computing

GPGPU technologies have attracted quite a lot of interests in research and application from many areas including databases. A set of GPGPU primitives to support relational operators have been developed on top of classic parallel algorithms including sorting and scan [24]. As part of the open source GDB release, a Cache Conscious B+-tree has been implemented on GPU to support indexed nest loop join, in addition to non-index nest loop join, sort-merge join and hash join [27]. More recently, Bakkum and Skadron [8] have implemented a subset of the SQLite command processor directly on GPU. While these pioneering works have demonstrated the effectiveness of speeding up read-only relational operations and have set a solid base for further investigations, it is unclear how they can be extended to indexing and querying raster geospatial data as relational data has quite different characteristics from raster geospatial data. More recently, GPU has been used to batch processing a large number of simultaneous queries on tree structures [31]. GPU has also been used to speed up similarity joins on point data using the Set of Z-Lists (SZL) data structure [32] which is based on space filling curve and is non-hierarchical. It is also unclear how the technique can be applied to raster geospatial data to process ROI-type query efficiently. We note that for the works reported in [58][9][20], while GPU was used to accelerate database operations, they were based on the graphics rendering pipelines and not GPGPU computing technologies.

Different from the works by [24] and [8] that try to build a complete database on GPU, in this study, we use GPU more as an accelerator for indexing in a way similar to the GPU's original role as an accelerator for graphics rendering. This is partially because quadtree indexing of large-scale raster geospatial data is computationally expensive and it is mostly one-time process for read-only data. On the other hand, processing window queries and ROI-type queries is much less expensive. Our previous work has shown that ROI-type queries can be processed in a fraction of a second for global 1-km resolution data with a 16-level quadtree using a single CPU core[73]. Another disadvantage of using GPU for indexed query processing is that the overheads of data transfer between CPU and GPU may overshadow the benefits of GPU based parallel query processing. In addition, compared with CPU, the GPU global memory is still very limited and can not hold large indices and/or data. Frequently swapping data/indices between CPU and GPU memory may not be desirable with respect to overall performance. As such, we will be focusing on indexing large-scale raster geospatial data on GPUs while leaving query processing on CPUs.

2.3 Indexing Raster Geospatial Data

There are relative few works on indexing raster geospatial data compared with indexing vector geospatial data. While interval tree [15], octree [68][63] and kd-tree [22][29] have been extensively used in 3D graphics such as iso-surface rendering and ray-tracing, quadtrees have been widely used to index binary and gray scale 2D rasters for compression purposes [53][33][12][14]. Data structures and algorithms designed for compression are not necessarily suitable for query processing. Pyramid and tiling techniques have also been used to speed up image display but usually they do not allow queries on the underlying raster data. Oracle GeoRaster [43]allows storing the bounding boxes and derived attributes of tile images as vector geospatial data, which subsequently can be indexed and queried so that only selected tile images need to be retrieved for display. A few of existing works have addressed the issue of managing a set of similar/related rasters based on overlapping quadtrees [62][34][35]. The techniques are similar to indexing spatial-temporal vector geospatial data such as Historical R-Tree [38], MV3R-Tree [59], TPR-Tree [51] from a methodology perspective. All the above indices construction algorithms are serial. It is desirable to investigate how modern GPU hardware devices and GPGPU parallel computing technologies can be effectively used to index large-scale raster geospatial data to support fast ROI-type queries.

Techniques such as linear quadtrees [52] have been developed to externalized main-memory based quadtrees and make them disk-resident. Linear quadtrees can be used to support certain types of queries on top of B+-Tree [62][1][34]. Dynamically loading disk-resident quadtrees into main-memory when they are needed certainly reduces memory requirement; however, converting linear quadtrees to pointer quadtrees may incur significant overheads. In our previous work on managing large-scale species distribution data [71], we have associated a set of species identifiers with linear quadtree nodes and have used the PostgreSQL LTREE module to perform window queries by coordinating both the query client and the database server. We have also developed a Binned Min-Max Quadtree (BMMQ-Tree) data structure that associates min/max values of raster cells of a quadrant to the corresponding quadtree node to speed up ROI-type query processing in a Web environment [73]. BMMQ-Tree is a main-memory data structure constructed through a recursive procedure and can not be easily adapted to GPU devices. In this research, we focus on developing a main-memory data structure with low memory footprint, and, more importantly, can be efficiently constructed on GPUs.

We note that ROI-type queries on a set of large-scale 2D rasters are quite different from iso-surface generation or ray-tracing for 3D rasters [63][29]. First, techniques designed for 3D rasters focus on tracing boundaries (iso-surfaces) and intersecting with linear objects (ray-tracing) while 2D ROI-type queries find spatially continuous regions that satisfy compound query criteria. Second, unlike 3D data, there is a mismatch between X/Y and Z dimensions for 2D rasters. At the 30 arc-seconds (approximately 1 kilometer) resolution, a global 2D raster data has a size of 43200 by 21600 while the cell sizes along the X and Y dimensions typically do not exceed 1024 by 1024 in 3D applications. Data structures that are suitable for high resolution 3D data (e.g., 1024*1024*1024) are not necessarily suitable for a set of large-scale 2D rasters.

2.4 Parallel Processing of Geospatial Data

Parallel and distributed processing of geospatial data is not a completely new concept. Quite a few works on parallel spatial data structures [13][30][28][2], spatial join [74][47], spatial clustering [70], spatial statistics [5][64] and handling terrain data [50] have been reported. However, as discussed in [16], research on parallel and distributed processing of geospatial data prior to 2003 has very little impact on mainstream geospatial data processing applications, possibly due to the accessibility of hardware and infrastructures in the past. Among these works, very few of them specifically addressed query processing for raster data. Cary et al [11] reported their experiences on processing spatial data with MapReduce on a Google and IBM cluster using the Hadoop framework [4]. Their experiments include R-Tree construction on point data and image tile quality computation. Parallel computing on LIDAR data using cluster computers [23] is getting increasingly popular due to its computation intensive nature. More recently, works reported in [65] have demonstrated significant speedups by using grid computing for spatial statistics. However, none of these works adopted GPGPU computing technologies.

3. THE PROPOSED SOLUTION

3.1 Array-Representation of CCQ-Tree

The Cache Concisions Quadtree (CCQ-Tree) we propose in this paper is motivated by the pioneering works on Cache Sensitive Search Tree (CSS-Tree) and Cache Sensitive B+-Tree (CSB+-Tree) by Rao and Ross [48][49]. CCQ-Tree uses an array representation and places all nodes in an one-dimensional array to completely remove non-continuous memory allocations. The layout of a CCQ-Tree node includes a user-defined data field and a field indicting the position of its first child on the node array. In both spatial window and ROI-type queries, all the child nodes of a tree node being examined need to be visited sequentially. As such, putting all its child nodes consecutively in an array instead of storing them in disparate memory addresses will improve cache hits [48][49]. Compared with classic main memory quadtrees storing memory pointers to four child nodes, storing only one position number to a node's first child reduces memory consumption significantly. This is especially true when a node's data field is small. Also CCQ tree allows user to define the lengths of the data field as well as the first child position field. While a pointer usually requires the length of a word, e.g., 64 bits for a 64 bits machine, a 32 bits first child position field can refer to 4 billion nodes which is sufficient in most cases. When the node size is fixed, CCQ allows flexible allocations of the two fields based on applications. For example, assuming the node size is fixed to 64 bits, CCQ has the flexibility to either designate 32 bits to the data field and 32 bits to the first child position filed or designate 40 bits to the data field while 24 bits to the first child position field.

The lower part of Fig. 1 illustrates the layout of a CCQ-Tree through an example. In this particular example, the data field of a tree node includes a minB component and a maxB component to store the minimum and maximum values of all the raster cells under the node. The root node's minB and maxB are 0 and 4, respectively and the first child position is 1. We can visit the second child of the root by adding the first child position of the root node (which is 1) and the offset (which is 1) and retrieve the node from the array at position 2. Empirical studies have shown that, while environmental data are well-known for significant spatial autocorrelation due to the first law of geography [61], i.e., "Everything is related to everything else, but near things are more related than distant things", neighboring cells values often are slightly different which makes traditional quadtree-based indexing that requires the uniformity of quadrants inappropriate. By binning cell values using proper boundary values, quadrant uniformity can be derived and the complexity of the derived quadtree can be reduced, including number of levels and memory footprint of the tree. The binning process is shown in the upper part of Fig. 1 using a predefined bin boundary lookup table.

3.2 Overview of CCQ-Tree Construction on GPGPUs

Constructing hierarchical data structures on GPUs imposes a major technical challenge compared to well-established recursive approaches on CPUs. Current GPGPU computing does not allow dynamic memory allocations and pointer referencing and dereferencing. Also, under the CUDA architecture, communications among blocks are very difficult if not impossible.

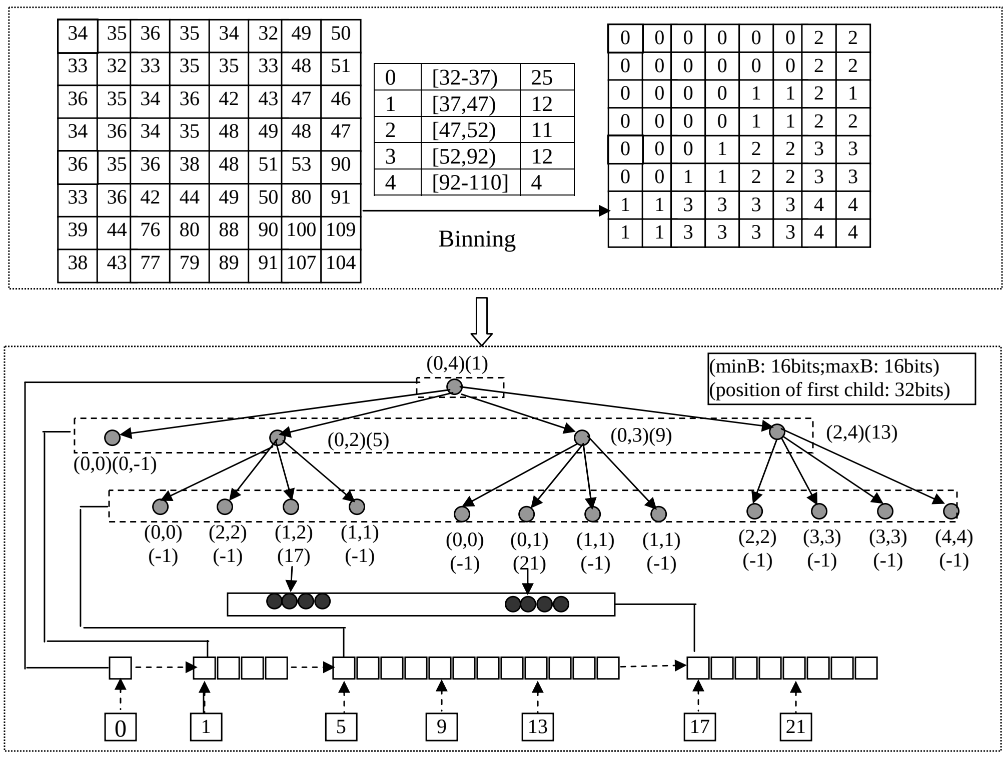
Recall that a CCQ-Tree node consists of a data field and a first child position field. We need to fill both fields of all the nodes to successfully construct a CCQ-Tree. In addition, since the nodes are processed in parallel, each node needs to know its position in the array representing the tree. The main idea of our CCQ-tree construction algorithm includes the following components: (1) build a pyramid of matrices from the raw raster data to materialize the data field in each node, (2) at each level of the pyramid, by checking whether an element of the matrix at the level has a valid value, it can be determined that whether the tree node corresponding to the element should be pruned. By traversing the pyramid from top to bottom and follow Morton-order [37] or Z-order [44] at the each level, the position of each tree node can be computed (3) At each level of the pyramid, by checking the number of valid children for each node, the first child position of a tree node can be determined using a similar approach.

Although the three steps are executed sequentially on CPU, quadrants of the raster are processed in parallel in all the three steps on GPU. The constructed tree in the form of an array of tree nodes will be finally transferred back to CPU and is ready for processing queries. Since the maximum of threads allowed by our GPU device (Nvidia Quadro FX3700) is 512, we have used a square layout of T*T ($T = 2^U$ where U<=4) threads per block. Assuming that our raster size is N*N ($N = 2^K$) and we use P*P blocks ($P = 2^L$) blocks, then each thread will be responsible for processing Q*Q elements ($Q = 2^{K-L-U}$) at the finest level. We will present the algorithm in details in the following subsections.

3.3 Parallel Building Data Pyramid

Given a raster R of size N*N where $N = 2^K$, the pyramid for the raster includes K-1 matrices of sizes N/2 by N/2, N/4 by N/4 ... and 1*1. The value of an element of matrix at the level k of the pyramid is a function of the four elements under it at the level k+1 (assuming the root has a level of 0), i.e., $d_k = f(d_i^{k+1}|i = 0, 3)$. We define function f as computing minimum and maximum values to make it comparable to our previous work on Binned Min-Max Quadtree (BMMQ-Tree) [73] although it can be defined differently depending on applications. In this case, each matrix element matrix has two fields, the minimum and the maximum. Note that while the pyramid does not include R, the bottom level matrix is computed from R by applying function f to the corresponding elements of R. It is not difficult to compute the total number of elements of the pyramid as $M = (\frac{1}{4}+\frac{1}{4^2}+...+\frac{1}{4^{K-1}})*N*N = \frac{N^2}{4}(\frac{1-(\frac{1}{4})^K}{1-\frac{1}{4}}) = \frac{N^2}{3}(1-(\frac{1}{4})^K)$. When K is reasonably large, M is approximately $N^2/3$, i.e., one third of the total size of R.

To build the pyramid on GPU, we allocate a one-dimensional array of size $N^2/3$ on CPU, initialize the array and transfer it to GPU. A kernel using P*P blocks for the whole data grid and 16*16 threads per block was adopted. When the kernel to populate the pyramid was invoked, as discussed before, each thread needs to process Q*Q elements at the level K-1 of raster R. All the matrices are generated bottom-up in parallel at each level. For the matrices between levels K-1 and L+U, the workload of each thread is reduced to 1/4 due to the aggregations of elements. At the level L+U, each thread will only process four elements of the level L+U+1 matrix. The sizes of matrices between level L and L+U are below the number of threads that are allocated to the previously defined kernel. To utilize the GPU device fully, we reduce the block size and



34	35	36	35	34	32	49	50
33	32	33	35	35	33	48	51
36	35	34	36	42	43	47	46
34	36	34	35	48	49	48	47
36	35	36	38	48	51	53	90
33	36	42	44	49	50	80	91
39	44	76	80	88	90	100	109
38	43	77	79	89	91	107	104

0	[32-37)	25
1	[37,47)	12
2	[47,52)	11
3	[52,92)	12
4	[92-110]	4

0	0	0	0	0	0	2	2
0	0	0	0	0	0	2	2
0	0	0	0	1	1	2	1
0	0	0	0	1	1	2	2
0	0	0	1	2	2	3	3
0	0	1	1	2	2	3	3
1	1	3	3	3	3	4	4
1	1	3	3	3	3	4	4

Figure 1: Illustration of CCQ-Tree Layout and Raw Data Binning

thread size per block gradually by lunching new kernels at each level. At the certain level S, we switch to serial execution either on CPU or use a single thread per SM, as the costs of launching new kernels may overshadow the parallel processing benefits. For large rasters, P*P ($P = 2^L$) may be quite larger than the number of blocks that can be accommodated by all the SMs at the same time, thus S is likely to be smaller than L.

3.4 Parallel Computing of First-Child Node Positions

As mentioned earlier, a multiple-level Z-order [44] based algorithm is proposed to compute the first child node positions that are required for all the non-leaf tree nodes in a derived CCQ-Tree. The algorithm to compute first child node positions has three steps. First, similar to the min/max pyramid (A), we also create a pyramid of matrices to record the number of children for each matrix elements of the pyramid (B). The computation can be performed by launching GPU kernels in a way similar to the method presented in the previous subsection. Also in this step, a prefix sum (scan) [26][10]is performed in parallel to accumulate the numbers of children for all elements with at least one child. The value of the last element in the matrix at every level is the total number of children for all the elements at the level. The numbers at the all levels are used to formulate an array (W) and subsequently a prefix sum is performed to compute the starting position of the first valid element at all the levels. As the number of levels is limited (typically <20), this step can be performed at CPU quickly. Note that the root takes one position and should be counted as shown in Fig. 3. The last step of the algorithm is to finally compute the first node positions of all elements that have at least one child. This is done by adding the starting position of the first valid element at the each level to all the matrix elements after applying prefix sum as in the first step. As discussed in the next subsection, we need the number of children pyramid (B) to generate a CCQ-Tree in the final step, it is not possible to perform in-place prefix sum. As such, we make a copy of B and use its value as the initial for the first child node position pyramid (C).

An issue unexplained in the algorithm given above is how to determine whether an array element in the matrices of pyramid A should be considered as a tree node. While different rules can be applied which may generate different trees, the following rule is used in this study: if an element has different minimum and maximum values or if any of its min/max values is different from its parent's min/max values, respectively, then the element is considered to be a tree node. To help illustrate the algorithm better, an example is presented in Fig. 3. As shown in the top part of Fig. 3, all the elements of A1 should be considered to be tree nodes as they have different min/max values and their min/max values are different from their parent's min/max values. In contrast, the top-left four elements of array A2 have the same min/max values and they are the same as their parent's min/max values. As such, they will be pruned from the quadtree. Based on the rule, we can derive B2 from R and A2, derive B1 from A2 and A1 and derive B0 from A1 and A0 by following step 1. The numbers of children at the three levels are thus 4, 12, 8, respectively. Since the root node takes a position, the numbers of children array will be W=(1,4, 12, 8) and thus the corresponding array after prefix-sum will be W=(0,1,5,17) after step 2, as shown in the middle of Fig. 3. To illustrate step 3, we use the derivation of C1 as an example. The initial value of

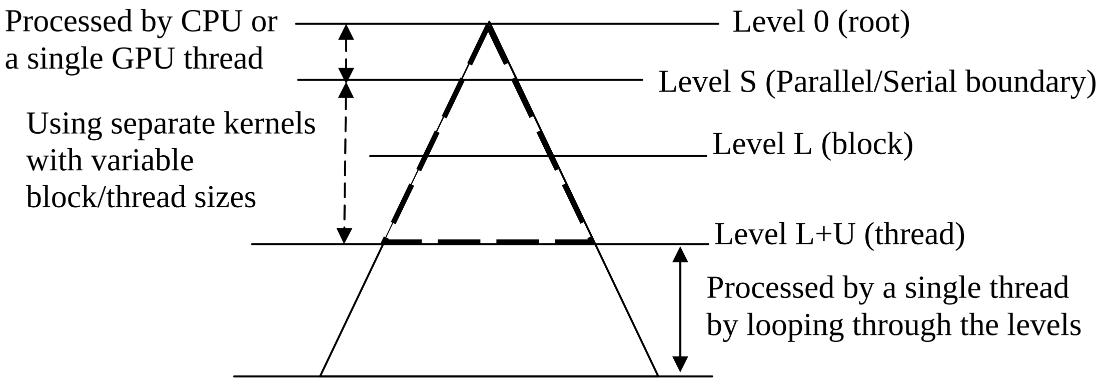

Processed by CPU or a single GPU thread
Level 0 (root)
Level S (Parallel/Serial boundary)
Using separate kernels with variable block/thread sizes
Level L (block)
Level L+U (thread)
Processed by a single thread by looping through the levels

Figure 2: Decomposition Schema for Data Pyramid on GPU

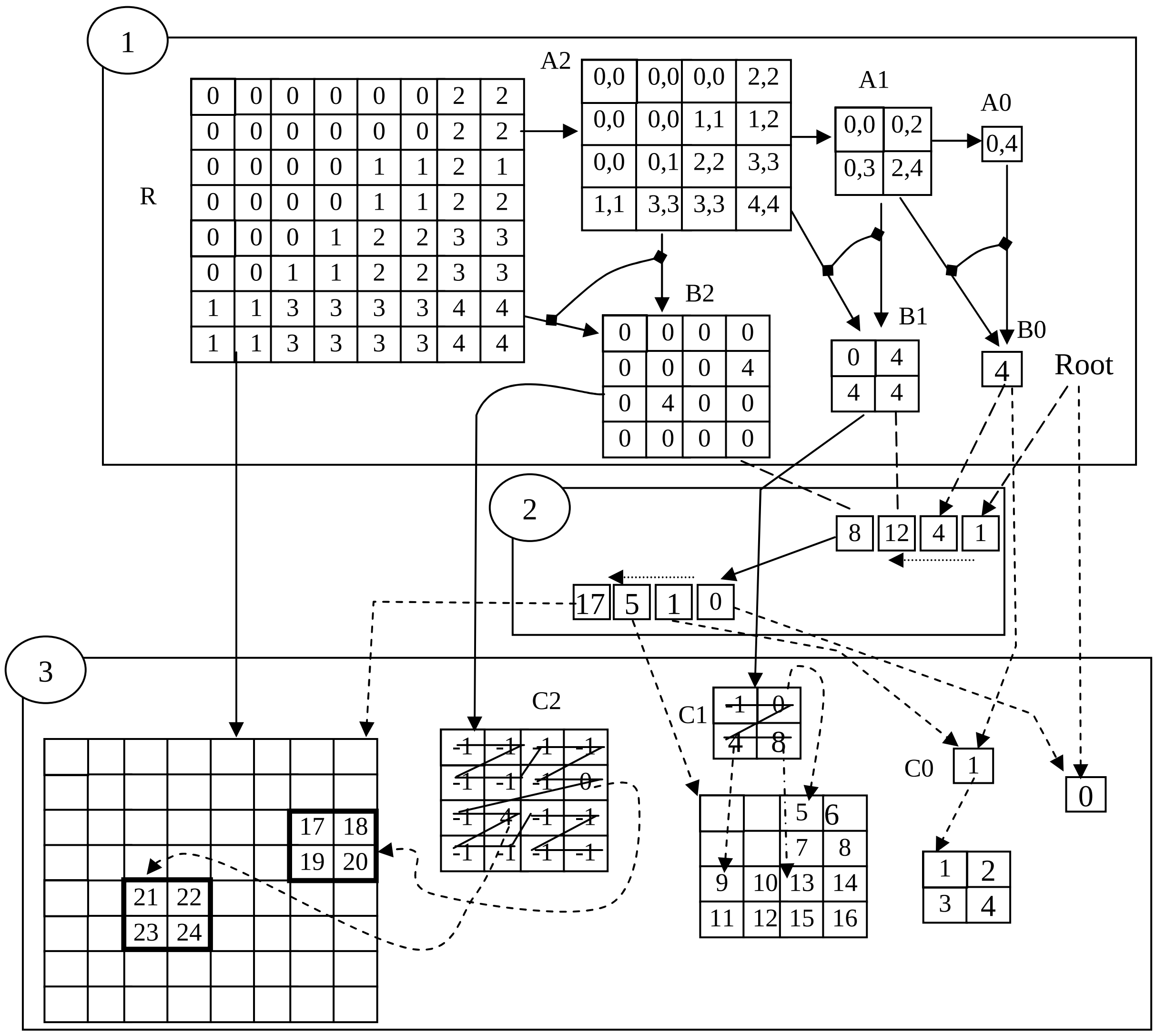


Figure 3: Illustration of Parallel Computing First-Child Node Positions

C1 is copied from B1, i.e., (0,4,4,4). After applying the prefix sum based on the Z-order, C1 becomes (0, 4, 8, 12). After adding the start position of the level, i.e., W[1]=1, to all the elements of C1 (except those whose values of the corresponding elements in B are 0), C1 becomes (-1, 5, 9, 13). Each element of C1 is the first child node position of the corresponding tree node being constructed. For example, the tree node corresponding to the second element of A1 has four children (B1[1]=4) and its first child node position is 5 (C1[1]=5). On the other hand, the tree node corresponding to the first element of A1 has 0 children (B1[0]=0) and its first child node position is set to -1 (C1[0]=-1).

3.5 Parallel Generating CCQ-Tree

After computing the first child node positions of all the valid matrix elements in pyramid C, we are ready to convert the pyramid representation into a compact one-dimensional array representation to reduce memory footprint. After the step 2 of the first child node position calculation algorithm, we are able to know the total number of nodes for the CCQ-Tree being generated (assuming S) which is given by the last element of the W array after prefix sum in the step. We thus allocate an array (E) of structures with size S on GPU. A major remaining question is, for each valid matrix element in the pyramid, how do we tell their positions in the array representation? The answer is to apply a similar technique described in the previous section. What we need to do is to replace the numbers of children with 0s and 1s based on whether the elements have at least one child. After Step 3 is finished, the matrix element values in the pyramid (D) will be the positions in the one-dimensional array of the corresponding matrix elements. Note that prefix sum on D can be done in-place.

Assuming the data pyramid is A, the number of child pyramid is B, the first child position pyramid is C, the node position pyramid is D and the CCQ-Tree array is E, then we can derive E from A, B, C and D on GPU in parallel as follows: using a similar block/thread layout as discussed in Section 3.2, for each matrix element in raster R's pyramid (A, B, C or D), the algorithm first checks its number of children using B, if the number is large than 0, then the element should be a node in the CCQ-Tree and should be put in array E at the correct position; the position of the node in array E can be retrieved from pyramid D, the data field can be copied from pyramid A and the first child node position can be retrieved from pyramid C, all from the corresponding matrix elements in the respective pyramid.

3.6 Discussions

By using a few auxiliary pyramids, we have successfully developed a highly parallel algorithm to generate a CCQ-Tree on GPUs. The algorithm only uses GPU's global memory and SIMT (Single-Instruction Multiple-Thread) parallel computing paradigm to achieve maximum interoperability. As discussed previously, the total number of matrix elements in each of the A, B, C, and D pyramids is approximately one third of the total size of R. Assuming R is the short integer type, the min/max value and the number of children can be packed into 24 bits and the node position and the first child position, each takes a 24 bit short integer, then the additional memory requirements is about the 3/2 of the raw data, as $\frac{1}{3}N^2*(24+24+24)/16 = \frac{3}{2}N^2$. We note that 24 bits for node positions can address 2^{24}=16M array elements which should be more than enough for a CCQ-Tree. Most likely the GPU global memory will be used up by the raw raster data and the auxiliary pyramids before a CCQ-Tree's number of nodes reaches such a level.

We are in the process of exploring Nvidia GPU devices' unique features to further improve system performance, such as shared memory, coalesced memory access and const/texture memories [40]. As global memory access is one of the most expensive operations (usually a few hundreds of cycles) and the most workload of the algorithm is at the lower levels where each thread needs to access a large number of raster cells and pyramid matrix elements, it is desirable to reduce memory access times as much as possible. Unlike caching in CPU, coalesced memory access in CUDA can only be shared by threads in the same execution group, not by the next execution of the same thread [40]. This makes it very tricky to utilize coalesced memory access in CUDA for Nvidia GPUs. Fortunately, the next generation Nvidia Fermi GPU architecture provides a unified L2 cache in addition to allowing users to allocate a part of the per-block based shared memory as L1 cache [41]. We plan to explore these new features when the Fermi-based GPU devices are available to us.

Another related issue under investigation is how to combine multiple non-overlapping CCQ-Trees into a bigger one. Due to the limited global memory on GPU devices and the extra memory consumptions incurred by the auxiliary pyramids, a single GPU can not process very high resolution raster data (e.g., 1 km) at the global scale. A possible solution is to generate smaller CCQ-Trees on multiple GPU devices or generate smaller CCQ-Trees on a single GPU with multiple runs and then combine them. We are investigating both possibilities.

4. EXPERIMENTS AND EVALUATION

To verify the correctness of the proposed CCQ-Tree construction algorithm on GPUs and tests its efficiency, we report our results on a real raster dataset. We compare the GPU based solution with single core CPU based one with respect to index construction time. We also compare memory footprints and index loading time of CCQ-Tree with that of classic pointer-based quadtrees as they are important to query processing.

4.1 Data and experiment setup

We use the current climate data published by WorldClim [25][69]. It is the same dataset that we have used for the experiments reported in our previous works [73][72] to allow direct comparisons as detailed in the next two subsections. For the sake of completeness, here we provide a brief description of this dataset. The WorldClim dataset is the interpolations of in-situ observed data from 1950-2000 and includes monthly precipitation, minimum temperature and maximum temperature (12 month) and 18 derived bioclimatic variables at the global 30 arc-seconds (1 kilometers) resolution. Thus the number of grid cells is 43200*21600 for each of 12*3+18=54 rasters. Due to space limit, we only report the experiment results using the January precipitation data. The minimum and maximum precipitation values are 0 and 1003, respectively. We set the maximum level of the CCQ-Tree to 16, i.e., K=16, as 2^{16}=65536 is already larger than 43200. While experiments using different bin numbers have been performed, we only report the results using 8 bins due to space limit. Clearly using larger number of bins results in more complex quadtrees but incur less false positive rates and there is a tradeoff between index memory footprint and disk IOs to remove the false positives.

The dataset is stored in BIL format and we use GDAL [19] to access the data file and read the data into main memory. We do not count disk I/O costs for data as our motivation is to use indices to minimize disk I/Os. With indices, unnecessary disk I/Os can be significantly reduced. Our Nvidia Quadro FX 3700 card is a fairly aged model and it only has 512M device memory available to GPGPU computing. As such, we are only able to generate CCQ-Trees for image tiles of 4096*4096 and there are 11*5 raster tiles in our experiments. Note that some of the tiles on the bottom or right boundaries are padded with NO-DATA values. Also some of the tiles that mainly cover oceans have very few cells with valid data.

4.2 Tests on Indices Construction Times

The CCQ-Tree construction times on GPU for the 55 4096*4096 image tiles are shown in Fig. 4. For easy comparisons, the quadtree construction times on CPU are also shown in Fig. 4. The minimum and maximum CCQ construction times are 0.38 second and 0.47 second, respectively, with an average of 0.42 second. In contrast, the quadtree construction on CPU takes 9.28 seconds at minimum and 10.11 seconds at maximum with an average of 9.83 seconds. The average speedup among the 55 tests is 23.4 times which is considerably significant.

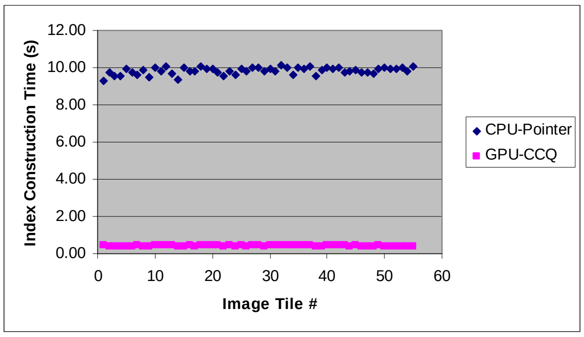


Figure 4: Comparison of Indices Construction Times on GPU and CPU

While it takes 23.02 seconds for the CCQ-Tree construction algorithm to index the global 1-km dataset with 43200*21600 cells, we expect that the construction time can be reduced to 1-5 seconds using Fermi based GPU cards for the following reasons. First, while FX3700 card has only 112 cores, Fermi cards have 480 cores which are four times more in addition to higher clock rate. Second, as indexing geospatial raster data is data intensive, accessing global memory is likely to be a major factor that affects indices construction times. As Fermi cards now support a unified 768K L2 Cache and per-SM L1 cache up to 48K, we expect the overall indices construction times to be significantly reduced by utilizing the caches which are not available to the FX3700 card that we are currently using. It is also possible to further reduce construction times by using multiple GPU cards. When the indices construction times for global 1km resolution data can be reduced to a few seconds, many traditional offline applications can then be performed online in an interactive manner which may potentially have significant implications in managing large-scale geospatial data.

4.3 Test on Memory Footprints and Indices Loading Times

As CCQ-Tree is designed to be memory-resident for the purpose of speedup query processing in a way similar to the BMMQ-Tree developed in our previous work [73], it is important to reduce memory footprint so that a limited main-memory capacity can accommodate larger CCQ-Trees for large-scale raster geospatial data. In addition, in our previous study, we have transformed BMMQ-Trees to linear quadtrees so that they can be permanently stored on disks. Unfortunately, we have to build BMMQ-Trees from the corresponding linear quadtrees when they are loaded into main-memory. In this test, we will compare the memory footprints and index loading times among BMMQ-Trees and CCQ-Trees.

In addition to the minimum and maximum values that are the same as in a CCQ-Tree, a BMMQ-Tree has four pointers to its four children instead of a number indicating the position of its first child. As such, for the sake of simplicity, assuming that the minimum and maximum values are 16 bits short type and the first child node position is 32 bit integer type, a BMMQ-Tree node will require 16+16+32*4=160 bits (20 bytes) while a CCQ-Tree node requires 16+16+32=64 bits (8 bytes), a 60% reduction. On a 64 bits machine, a BMMQ-Tree node will be 16+16+64*4=298 bits (36 bytes) while the size of a CCQ-Tree node remains unchanged (8 bytes) which indicates 77% reduction.

The speedups of indices loading times for BMMQ-Trees over CCQ-Trees are plotted in Fig. 5 against the number of tree nodes. As we can see from Fig. 5, the speedups of loading times quickly reaches 50 to 90 when the numbers of tree nodes are above 5000. This is not surprising as loading a BMMQ-Tree requires examining all the bits of the Morton code of the corresponding liner quadtree node, dynamically allocating memory and set pointers correctly. On the other hand, it only takes a single read to copy the array representing a CCQ-Tree into memory.

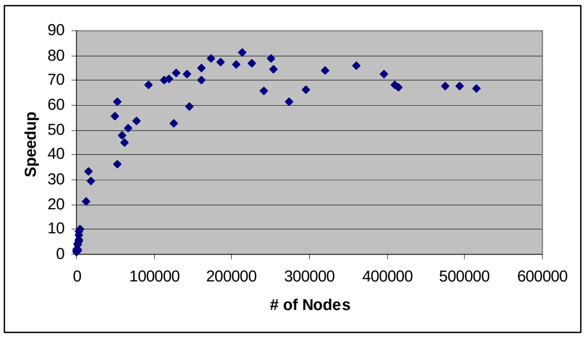


Figure 5: Plot of Speedup of Indices Loading Times against # of Tree Nodes

5. CONCLUSIONS AND ONGOING WORK

In this study, we reported our work on indexing large-scale geospatial raster using massively parallel GPGPU computing. Towards this end, we have designed the CCQ-Tree data structure which is not only memory efficient but also suitable for GPU-based indexing. Using an Nvidia Quadro FX3700 graphics card, we are able to improve tree indices construction times from 9.28-10.11 seconds on a CPU core to 0.38-0.47 second with an average speedup of 23 times. Compared with CPU main-memory pointer quadtrees, CCQ-Tree not only reduces memory footprint to 40% and 23% on a 32-bits and 64-bits machines but also speeds up indices loading times 50X-90X by eliminating the expensive linear quadtree to pointer quadtree conversion costs. While currently the total indices construction time for a global 1-km spatial resolution dataset takes 23.02 seconds on the FX3700 card, we project that a personal workstation equipped with 1-4 Fermi GPU cards can index global 1-km spatial resolution datasets in a few seconds. The capability of indexing large-scale high resolution datasets in real time can potentially have significant implications in managing geospatial data.

For the future work, first of all, we would like to fine-tune our CCQ-Tree data structure and its construction algorithm for the Nvidia Fermi GPU architecture to further improve their performance. Second, we are interested in developing a complete set of high-performance computing primitives on mini-clusters equipped with GPU cards for visual explorations of large-scale complex geospatial datasets, such as identifying global biodiversity patterns and their relationships with the environment. Finally we would like to compare our GPU computing based solution with traditional grid/cluster computing based ones in the context of indices construction, query processing and visual explorations of large-scale geospatial data.

6. REFERENCES

[1] A. Aboulnaga and W. G. Aref. Window query processing in linear quadtrees. *Distributed and Parallel Databases*, 10(2):111–126, 2001.

[2] M. H. Ali, A. A. Saad, and M. A. Ismail. The pn-tree: A parallel and distributed multidimensional index. *Distributed and Parallel Databases*, 17(2):111–133, 2005.

[3] AMD/ATI. Ati stream technology. `http://www.amd.com/stream`.

[4] Apache. Hadoop. `http://hadoop.apache.org/`.

[5] M. P. Armstrong, C. E. Pavlik, and R. Marciano. Parallel-processing of spatial statistics. *Computers & Geosciences*, 20(2):91–104, 1994.

[6] K. Asanovic and R. B. et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, UC Berkeley, December 18 2006.

[7] ATI. Ati radeon. `http://en.wikipedia.org/wiki/Radeon`.

[8] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[9] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB'04*, pages 1021–1032, 2004.

[10] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[11] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM'09*, pages 302–319. Springer Lecutre Notes in Computer Science (LNCS), 2009.

[12] Y. K. Chan and C. C. Chang. Block image retrieval based on a compressed linear quadtree. *Image and Vision Computing*, 22(5):391–397, 2004.

[13] C. H. Chien and T. Kanade. Distributed quadtree processing. In *Proceedings of the first symposium on Design and implementation of large spatial databases*, pages 213 – 232, 1990.

[14] K. L. Chung, Y. W. Liu, and W. M. Yan. A hybrid gray image representation using spatial- and dct-based approach with application to moment computation. *Journal of Visual Communication and Image Representation*, 17(6):1209–1226, 2006.

[15] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE TVCG*, 3(2):158–170, 1997.

[16] A. Clematis, M. Mineter, and R. Marciano. High performance computing with geographical data. *Parallel Computing*, 29(10):1275–1279, 2003.

[17] B. Dally. The future of gpu computing. `http://www.nvidia.com/content/GTC/documents/SC09_Dally.pdf`.

[18] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[19] GDAL. Geospatial data abstraction library. `http://www.gdal.org/`.

[20] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD'04*, pages 215–226, 2004.

[21] GPGPU. General purpose graphics processing unit. `http://gpgpu.org/`.

[22] A. Gress and R. Klein. Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. *Graphical Models*, 66(6):370–397, 2004.

[23] S. H. Han, J. Heo, H. G. Sohn, and K. Yu. Parallel processing method for airborne laser scanning data using a pc cluster and a virtual grid. *Sensors*, 9(4):2555–2573, 2009.

[24] B. S. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems*, 34(4), 2009.

[25] R. J. Hijmans, S. E. Cameron, J. L. Parra, P. G. Jones, and A. Jarvis. Very high resolution interpolated climate surfaces for global land areas. *International Journal of Climatology*, 25(15):1965–1978, 2005.

[26] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[27] HKUST. Gpuqp: Query co-processing using graphics processors. `http://www.cse.ust.hk/gpuqp/gdb.zip`.

[28] E. G. Hoel and H. Samet. Data-parallel polygonization. *Parallel Computing*, 29(10):1381–1401, 2003.

[29] D. M. Hughes and I. S. Lim. Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on gpus. *IEEE TVCG*, 15(6):1555–1562, 2009.

[30] I. Kamel and C. Faloutsos. Parallel r-trees. In *SIGMOD'92*, pages 195–204, 1992.

[31] 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 *SIGMOD'10*, pages 339–350, 2010.

[32] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE'08*, pages 1111–1120, 2008.

[33] T. W. Lin. Compressed quadtree representations for storing similar images. *Image and Vision Computing*, 15(11):833–843, 1997.

[34] Y. Manolopoulos, E. Nardelli, G. Proietti, and E. Tousidou. A generalized comparison of linear representations of thematic layers. *Data & Knowledge Engineering*, 37(1):1–23, 2001.

[35] M. Manouvrier, M. Rukoz, and G. Jomier. Quadtree representations for storage and manipulation of clusters of images. *Image and Vision Computing*, 20(7):513–527, 2002.

[36] F. Molnar, T. Szakaly, R. Meszaros, and I. Lagzi. Air pollution modelling using a graphics processing unit with cuda. *Computer Physics Communications*, 181(1):105–112, 2010.

[37] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM

Ltd., 1966.

[38] M. A. Nascimento and J. R. O. Silva. Towards historical r-trees. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 235 – 240, 1998.

[39] Nvidia. Compute unified device architecture (cuda). `http://www.nvidia.com/object/cuda_home_new.html`.

[40] Nvidia. Cuda programming guide. `http://developer.nvidia.com/object/gpucomputing.html`.

[41] Nvidia. Next generation cuda architecture, code named fermi. `http://www.nvidia.com/object/fermi_architecture.html`.

[42] Nvidia. Personal super computing. `http://www.nvidia.com/object/personal_supercomputing.html`.

[43] Oracle. Georaster. `http://download.oracle.com/docs/html/B10827-01/geor-intro.htm`.

[44] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD'86*, pages 326–336, 1986.

[45] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[46] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. Times Cited: 161.

[47] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *ACMGIS'00*, pages 54–61, 2000.

[48] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99*, pages 78–89, 1999.

[49] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD 2000 Conference*, pages 475–486. ACM, 2000.

[50] D. K. D. Rokos and M. P. Armstrong. Experiments in the identification of terrain features using a pc-based parallel computer. *Photogrammetric Engineering and Remote Sensing*, 64(2):135–142, 1998.

[51] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD '00*, pages 331–342, 2000.

[52] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[53] H. Samet. Data-structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985. Times Cited: 25.

[54] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[55] R. R. Sinha, M. Winslett, and K. Wu. Finding regions of interest in large scientific datasets. In *SSDBM'09*, pages 130–147, 2009.

[56] I. P. Stewart. Quadtrees - storage and scan conversion. *Computer Journal*, 29(1):60–75, 1986.

[57] S. H. e. a. Stone, S. Accelerating advanced mri reconstructions on gpus. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.

[58] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD'03*, pages 455–466, 2003.

[59] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB'01*, pages 431 – 440, 2001.

[60] D. M. Theobald. *GIS Concepts and ArcGIS Methods,2nd Ed.* Conservation Planning Technologies, Inc, 2005.

[61] W. Tobler. A computer model simulating urban growth in the detroit region. *Economic Geography*, 46(2):234–240, 1970.

[62] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *ACMGIS'98*, pages 1–7, 1998.

[63] C. Wang and Y. J. Chiang. Isosurface extraction and view-dependent filtering from time-varying fields using persistent time-octree (ptot). *IEEE TVCG*, 15(6):1367–1374, 2009.

[64] S. W. Wang and M. P. Armstrong. A quadtree approach to domain decomposition for spatial interpolation in grid computing environments. *Parallel Computing*, 29(10):1481–1504, 2003.

[65] S. W. Wang, M. K. Cowles, and M. P. Armstrong. Grid computing of spatial statistics: using the teragrid for g(i)*(d) analysis. *Concurrency and Computation-Practice & Experience*, 20(14):1697–1720, 2008.

[66] Wikipedia. Nvidia geforce. `http://en.wikipedia.org/wiki/GeForce`.

[67] Wikipedia. Nvidia geforce 400 series specification. `http://en.wikipedia.org/wiki/GeForce_400_Series`.

[68] J. Wilhelms and A. Vangelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[69] WorldClim. Worldclim current conditions data 1950-2000. `http://www.worldclim.org/current`.

[70] X. W. Xu, J. Jager, and H. P. Kriegel. A fast parallel clustering algorithm for large spatial databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, 1999.

[71] J. Zhang, M. Gertz, and L. Gruenwald. Efficiently managing large-scale raster species distribution data in postgresql. In *ACMGIS'09*, pages 316–325, 2009.

[72] J. Zhang and S. You. Dynamic tiled map services: Supporting query-based visualization of large-scale raster geospatial data. In *Com.Geo'10*, page To Appear, 2010.

[73] J. Zhang and S. You. Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. In *SSDBM'10*, pages 379–396, 2010.

[74] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *GeoInformatica*, 2(2):175–204, 1998.