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

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

Simin You
Dept. of Computer Science
CUNY Graduate Center
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 quadtree data structure (CCQ-Tree) that is suitable for GPU indexing. A set of algorithms have been developed and integrated to construct CCQ-Trees on GPUs 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 a 112-core Nvidia Quadro FX3700 GPU device reduces construction times from around 9.83 seconds to 0.42 seconds (23X speedup).

Categories and Subject Descriptors
H.2 [Database Systems]

Keywords
Indexing, Raster, Large-Scale, GPGPU, Parallel Computing

1. INTRODUCTION
Advances in geospatial technologies have generated large amounts of raster geospatial data at ever increasing speeds. Different from vector geospatial data that sophisticated indexing techniques have been extensively developed and applied, research on indexing raster geospatial data is limited. 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/Earth). Massively parallel General Purpose Graphics Processing Unit (GPGPU) technologies [1], which allow using graphics processing units for general purpose computing, have provided personal computers with tremendous computing capabilities. For example, a single Nvidia Fermi-based GeoForce GTX480 GPU device has 480 cores and a peak floating point performance of 1.35 Terra Flops [2] at the cost of a few hundreds of dollars. The computing power provided by GPGPU enabled devices will enable many traditionally offline applications to run online and interact with users.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ACM GIS '10, November 2-5, 2010. San Jose, CA, USA (c) 2010 ACM ISBN 978-1-4503-0428-3/10/11...$10.00

In this study, we aim at fast indexing of large-scale raster geospatial data on GPGPU enabled devices to support Region-of-Interests (ROI) type of queries. A ROI-type query returns all spatial objects (including quadrants) that satisfy one or more value range criteria, e.g., temperature in the range [t1,t2) and precipitation in [p1,p2). We consider indexing raster geospatial data as a special global operation in processing geospatial data which is technically more challenging than parallelizing some local and focal operations on raster data. More specifically, we focus on quadtree based indexing due to its well-known data compression and pruning power in query processing [3]. 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 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. Our technical contributions can be summarized as follows:

- We design a cache conscious quadtree data structure (CCQ-Tree) that is suitable for GPU indexing.
- 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 a 112-core Nvidia Quadro FX3700 GPU card.

The remainder of the paper is structured as follows. Section 2 presents the CCQ data structure and its construction algorithm on GPUs. Section 3 reports our experiments on a real global dataset before concluding the paper in Section 4.

2. PROPOSED SOLUTIONS

2.1 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) [4][5]. CCQ-Tree uses an array representation and places all nodes in a 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 indicating the position of its first child on the node array. In 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. Compared with classic main memory quadtrees that store memory pointers to four child nodes, storing only one position number to a node's first child reduces memory consumption significantly. The lower part of Fig. 1 illustrates the layout of a CCQ-Tree through an example.

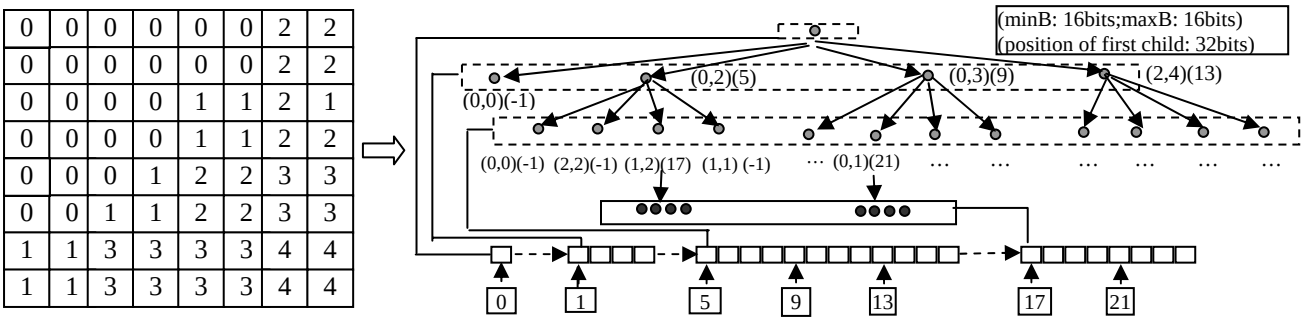

Fig. 1 Illustration of CCQ-Tree Layout

2.2 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. Recall that a CCQ-Tree node has 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 following Morton-order [6] or Z-order [7] at 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. We refer to [1] for more details on CUDA-based GPU computing. Assuming that a raster dataset has a size of N*N ($N=2^K$) and P*P computing blocks ($P=2^L$) are used in CUDA programming, 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.

2.3 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 the 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 0), i.e., $d^k=f(\{d_i^{k+1}|i=0,3\})$. We define f as a function to compute the minimum and maximum values to make it comparable to our previous work on Binned Min-Max Quadtree (BMMQ-Tree) [8] although it can be defined differently depending on applications. In this case, each matrix element 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 $M=(\frac{1}{4}+\frac{1}{4^2}+...+\frac{1}{4^{K-1}})*N*N=\frac{1}{4}*\frac{1-(1/4)^k}{1-(1/4)}*N^2=\frac{N^2}{3}*(1-(\frac{1}{4})^K)$. 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. When the kernel to populate the pyramid was invoked, each thread processes Q*Q elements at the level K-1 of raster R as discussed before. All the matrices are generated bottom-up in parallel at each level. For the matrices above level K-1, the workload of each thread is reduced to 1/4 due to the aggregations of elements.

2.4 Parallel Computing of First-Child Node Positions

A multiple-level Z-order [7] based algorithm is applied 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) [9][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 all the levels are used to formulate an array 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. 2. The last step of the algorithm is to finally compute the first node positions of all the elements that have at least one child. This is done by adding the starting position of the first valid element at each level to all the matrix elements by 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. Since it is not possible to perform an in-place prefix sum, we make a copy of B and use B's value as the initial for the first child node position pyramid (C).

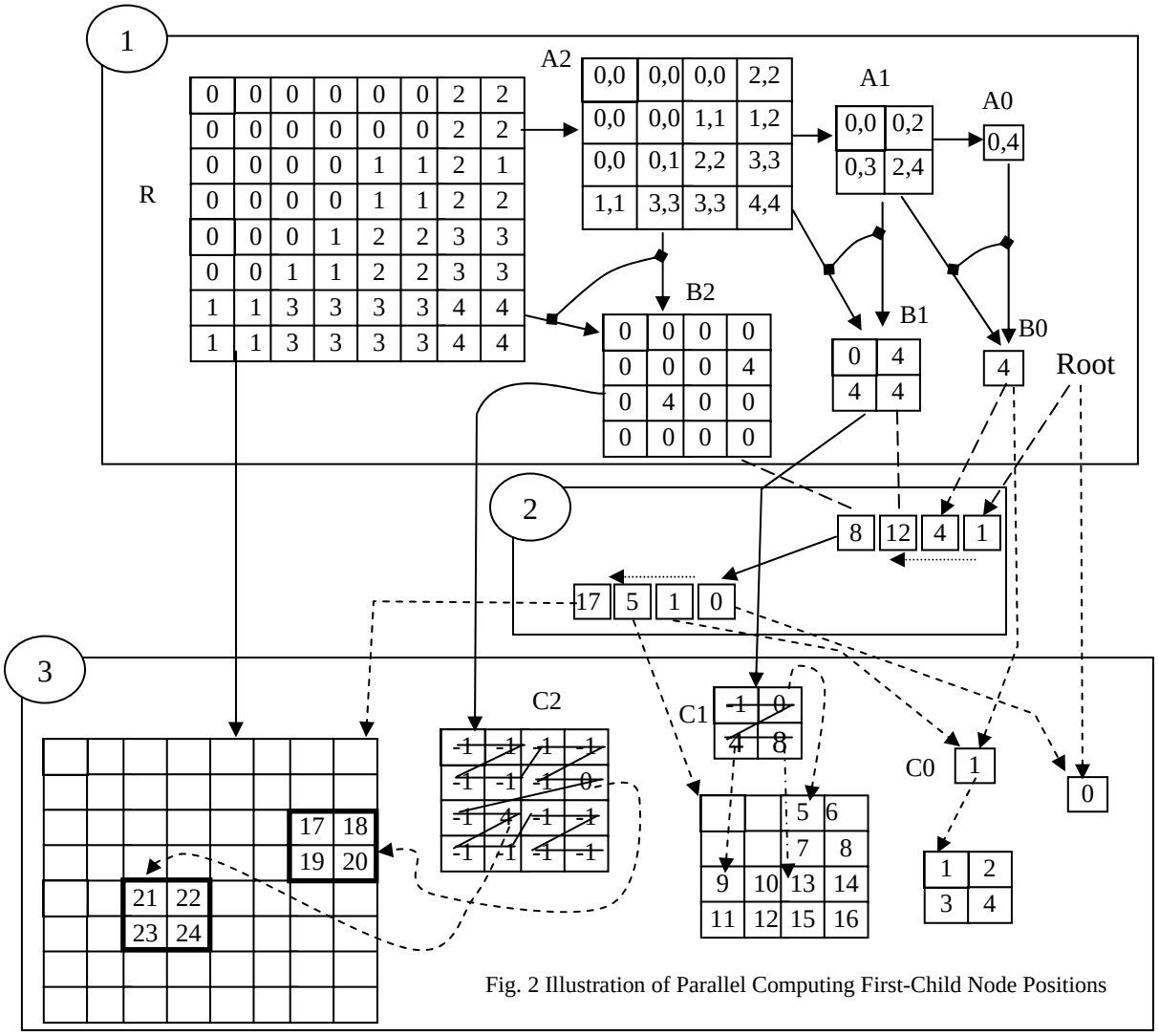

Fig. 2 Illustration of Parallel Computing First-Child Node Positions

An issue that has not been addressed 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. 2. As shown in the top part of Fig. 2, all the elements of A1 should be considered to be tree nodes as they have different min/max values or 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, and 8, respectively. Since the root node takes a position, the numbers of children arrays will be (1,4, 12, 8) and thus the corresponding array after the prefix-sum will be W=(0,1,5,17) after Step 2, as shown in the middle of Fig. 2. To illustrate Step 3, we use the derivation of C1 as an example. The initial value of C1 is copied from B1, i.e., (0, 4, 4, 4). After applying the prefix sum based on the Z-order, C1 becomes (0, 0, 4, 8). After adding the start position of the level, i.e., W[2]=5, to all the elements of C1 (except those of which the 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).

2.5 Parallel Generating CCQ-Tree

After computing the first child node positions of all the 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 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 array after

the prefix sum in the step. We thus allocate an array (E) of structures with size S on GPU. A major remaining question is that, for each valid matrix element in the pyramid, how do we tell its position 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 the 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 2.2, 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 EXPERIMENTS

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 index construction times incurred using the GPU based solution running on a 112-core Quadro FX 3700 card (500 MHz) and that using the single core CPU based one running on an Intel E5405 processor (2.0 GHz) that comes with a Dell T5400 workstation. We use the current climate data published by WorldClim [11][12] in our experiments. The dataset is the same one that we have used for the experiments reported in our previous works [8][13] to allow direct comparisons. The Nvidia Quadro FX 3700 card has only 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.

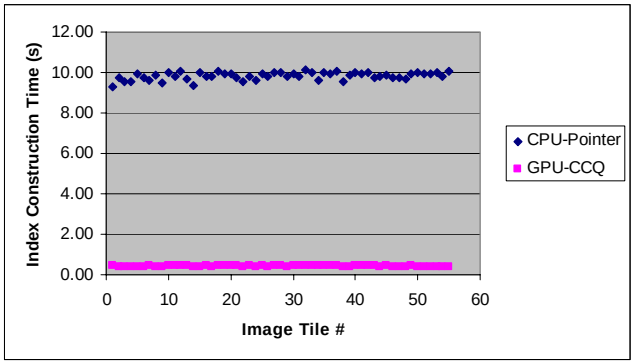

Fig. 3 Comparison of Indices Construction Times on GPU and CPU

The CCQ-Tree construction times on GPU for the 55 4096*4096 image tiles are shown in Fig. 3. For easy comparisons, the quadtree construction times on CPU are also shown in Fig. 3. The minimum and maximum CCQ construction times on GPU 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.

3. SUMMARY AND CONCLUSIONS

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 that is suitable for GPU-based indexing. Using a 112-core 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.

While currently the total indices construction time for a global 1-km spatial resolution dataset takes 23.02 seconds on a 112-core FX3700 card, we project that a personal workstation equipped with 1-4 Fermi GPU cards can index global 30-arc seconds spatial resolution (approximately 1km) datasets (43200*21600) in a few seconds. The capability of indexing large-scale high resolution datasets in real time can potentially have significant implications in managing and processing large-scale raster geospatial data.

REFERENCES

1. D. B. Kirk and W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
2. Wikipedia. Nvidia GeForce 400 series specification. http://en.wikipedia.org/wiki/GeForce_400_Series.
3. H. Samet. Foundations of Multidimensional and Metric Data. Structures. Morgan., 2005.
4. J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. VLDB'99, 78-89, 1999.
5. J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. SIGMOD'00, 475-486.
6. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., 1966.
7. J. A. Orenstein. Spatial query processing in an object-oriented database system. SIGMOD'86, 326-336, 1986.
8. J. Zhang and S. You. Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. SSDBM'10, 379-396, 2010.
9. W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. Communications of the ACM, 29(12):1170-1183, 1986.
10. G. E. Blelloch. Vector Models for Data-Parallel Computing. MIT Press, 1990.
11. 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.
12. WorldClim. Worldclim current conditions data 1950-2000.http://www.worldclim.org/current.
13. J. Zhang and S. You. Dynamic tiled map services: Supporting query-based visualization of large-scale raster geospatial data. Com.Geo'10, 2010.