Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs

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, 10016
syou@gc.cuny.edu

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

ABSTRACT

Global remote sensing and large-scale environmental modeling have generated huge amounts of raster geospatial data. While the inherent data parallelism of large-scale raster geospatial data allows straightforward coarse-grained parallelization at the chunk level on CPUs, it is largely unclear how to effectively exploit such data parallelism on massively parallel General Purpose Graphics Processing Units (GPGPUs) that require fine-grained parallelization. In this study, we have developed an efficient spatial data structure called BQ-Tree to code raster geospatial data by exploiting the uniform distributions of quadrants of bitmaps at the bitplanes of a raster. A fine-grained parallelization scheme has been implemented using Nvidia CUDA. Experiments show that the GPGPU implementation is capable of decoding a BQ-Tree encoded 16-bits NASA MODIS geospatial raster with 22,658*15,586 cells in 190 milliseconds, i.e., 1.86 billion cells per second, on an Nvidia C2050 GPU card. The performance achieves a 5.9X speedup when compared with the best dual quadcore CPU implementation and a 36.9X speedup compared with a highly optimized single core CPU implementation.

Categories and Subject Descriptors

H.2 [Database Systems]

Keywords

Bitplane, Bitmap, Indexing, Compression, Raster, Large-Scale, GPGPU, Parallel Computing

1. INTRODUCTION

High resolution large-scale raster geospatial datasets provide tremendous opportunities to understand the Earth and our environments deeper than ever before. Modern computing devices increasingly rely on parallel hardware architectures to meet the ever increasing demands of data processing power. Multicore CPUs and General Purpose Graphics Processing Units (GPGPUs) are the two leading hardware architectures that are already available in commodity computers. The data parallel nature of large-scale raster geospatial data matches these parallel hardware architectures very well. To make full use of the parallel computing capabilities, it is crucial to understand how spatial data structures and algorithms perform on these hardware architectures. Among numerous spatial data structures that have been proposed over the past thirty years, quadtree probably is the most popular family due to its effectiveness and simplicity in indexing, compressing and querying both vector and raster geospatial data [1-4]. While traditionally spatial data structures and algorithms assume uniform access cost to memory, the increasing performance gaps between different levels of memory hierarchy have made cache-conscious data structures significantly faster than their peers. However, the performance of classic spatial data structures (such as quadtrees) on modern commodity parallel processors (especially GPGPUs) is largely unknown.

Similar to bitmap based indexing in relational database, geospatial rasters can be transformed into a collection of bitplane bitmaps that are more suitable for indexing, compression and query processing. Among various operations on quadtree coded bitplane bitmaps (or bitplane quadtrees for short), encoding rasters into bitplane quadtrees and decoding bitplane quadtrees to restore original rasters (or decoding) are two fundamental operations. As encoding is a one-time task and usually can be done offline, it is technically more challenging to develop fast online parallel algorithms to decode bitplane quadtrees. Our focus in this study is to investigate the effectiveness of utilizing GPGPUs available in commodity personal computers for decoding bitplane quadtrees. The work is the first step towards developing a high-performance Geographical Information System (GIS) in a personal computing environment that allows Query Driven Visual Explorations (QDVE) of high-resolution, time-evolving and multi-variant raster geospatial data to effectively support global environmental studies.

2. BACKGROUND AND MOTIVATIONS

Raster data representation is a major data model for geospatial data. Surprisingly, compared to vector geospatial data that hundreds of indexing techniques have been developed, raster geospatial data is much less well supported in spatial databases with respect to efficient indexing and query processing. Existing techniques in spatial databases adopt a chunking approach to store raster geospatial data and index the metadata of the chunks using standard vector spatial indexing. While queries on the spatial locations and metadata values of the chunks are supported, chunks are stored as Binary Large Objects (BLOBs) with or without compression and usually no queries on the chunks are supported. The open source SciDB project [5] provides a comprehensive framework to manage multidimensional arrays, including raster geospatial data. While the current implementation does support generic compression methods, currently it does not support efficient queries on compressed chunks, i.e., compression is strictly for storage and does not benefit query processing. As quadtrees support raster compression and indexing simultaneously, we consider quadtrees a better choice for managing and querying large-scale raster geospatial data. However, classic quadtrees usually have overwhelming pointer

ACM SIGSPATIAL GIS '11, November 1-4, 2011. Chicago, IL, USA

(4/8 bytes) to data (1 bit) ratio when applied to bitplane bitmaps of rasters.

Bitmap indexing has been extensively investigated in relational databases. Unlike quadtree based query processing that naturally returns spatial hierarchy of resulting raster cells, queries based on classic bitmap indexing can only return individual tuples (correspond to raster cells) identifiers while the spatial relationships among the raster cells are lost. The respective advantages and disadvantages of quadtree based and bitmap based indexing have motivated us to develop a quadtree based efficient spatial data structure (BQ-Tree) to code bitplane bitmaps of large-scale raster geospatial data. Bitmap indexing has been widely used in commercial relational database systems and open source implementations (e.g., FastBit [6]) are available. BQ-Tree coding of raster geospatial data can reuse the bitmap based query processing framework and existing software codebase for fast system prototyping and practical environmental applications.

Our plan is to replace existing bitmap compression techniques such as run-length and Word-Aligned Hybrid (WAH) [7] that are spatial agnostic and utilize flat data structures, with the BQ-Tree coding to efficiently support both spatial (point, window, join) and attribute-based queries (exact, range, interval) on encoded geospatial rasters. While it is quite possible to directly perform queries on the BQ-Trees both serially and in parallel (which is left for future work), in this study, we adopt a simpler and more practical approach by parallel decoding BQ-Trees into bitmaps before executing queries. Query optimizers can choose to access only a subset of BQ-Trees that are relevant to a query to reduce I/Os. To process queries that require reconstructing raster chunks from encoded bitplane bitmaps, the BQ-Tree encoding is also beneficial as encoded bitplane bitmaps are usually much smaller than the raw raster chunks and thus expensive I/Os can be reduced. In this study, we focus on speeding up the reconstructions using massively parallel GPGPUs based on Nvidia Compute Unified Device Architecture (CUDA) [8].

3. THE BQ-TREE DATA STRUCTURE

Given a bitplane bitmap of a raster R of size N*N (assuming $N=2^n$), it can be represented as a quadtree where black leaf nodes represent quadrants of presence ("1"), white leaf nodes represent quadrants of absence ("0") and internal nodes are colored as gray. The quadtree can be easily implemented in main-memory by using pointers or stored on hard drives as a collection of linear quadtree paths. However, while the storage overheads of pointers or the paths can be justified if the length of the data field is much larger than the length of the pointer field (4 bytes for 32-bit machine and 8 bytes for 64-bits machine), the overhead is unacceptable as the data field is intended to be only 1-bit long to encode a bitplane bitmap. Furthermore, as the memory pointers are allocated dynamically and can point to arbitrary memory addresses, they are known to be cache unfriendly. To overcome these problems, we have designed a spatial data structure called BQ-Tree to efficiently represent bitmaps of bitplanes of a geospatial raster.

The basic idea of BQ-Tree is to sequence nodes of a regular quadtree into a byte-stream through breadth-first traversals with sibling nodes following the Z-order (Fig. 1). Different from classic main-memory quadtrees that use pointers to address child nodes, the child node positions in a BQ-Tree do not need to be stored explicitly. As such, the pointer field in regular quadtrees can be eliminated which reduces storage overhead significantly. In addition to tree nodes, a BQ-Tree also includes a compacted "last level" quadrant signature array. The layout of BQ-Tree nodes is as follows. Each BQ-Tree node is represented as a byte (8 bits) with each child quadrant takes two bits. We term the two bits as child node signature. The three combinations correspond to three types of nodes in classic quadtrees: "00" corresponds to white leaf nodes, "10" corresponds to black leaf nodes and "01" corresponds to gray nodes. The combination of "11" is currently not used. Child nodes corresponding to the quadrants with "00" or "10" signatures in their parent node can be safely removed from the byte stream as all the four quadrants in the child nodes are the same and their presence/absence information has already been represented in the respective quadrant signatures of the parent nodes. By consolidating four child quadrants' information into a single node, the depth of a BQ-Tree can be reduced by 1 when compared with classic quadtrees. The technique can potentially reduce memory footprint to up to 1/4.

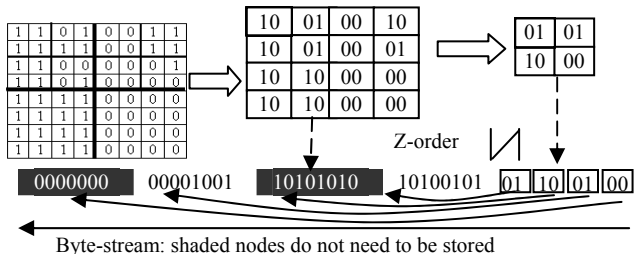

Fig. 1 Streaming BQ-Tree Nodes

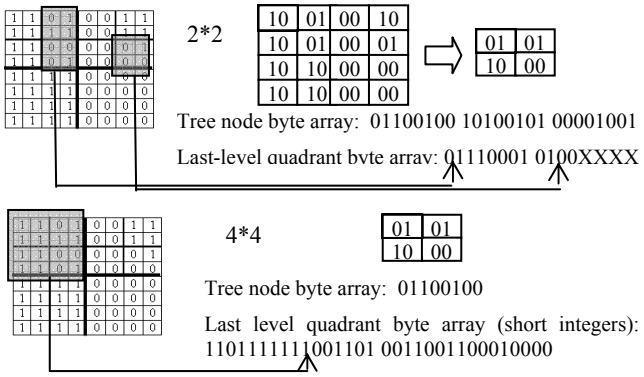

Fig. 2 Generating LLQS Array Using Different Quadrant Sizes

If we represent the four (2*2) raster cells in a quadrant as a BQ-Tree leaf node, then the second bit of the four quadrant signatures in the node will always be 0 (i.e., the signatures are either "00" or "10"). The redundancy is undesirable. To further reduce the memory footprint of the BQ-Tree for a bitplane bitmap, we introduce the concept of "Last Level Quadrant Signature", or LLQS. A last level quadrant is defined as a bitmap quadrant that is indexed by a 2-bit child node signature of a BQ-Tree leaf node. For the last level quadrant size of 2^k*2^k, we term the concatenation of the bits of the 2^k*2^k quadrant following a row-major order as the Last Level Quadrant Signature (LLQS). The LLQSs need to be recorded for the bitmap quadrants corresponding to BQ-Tree leaf node quadrants whose signatures are neither "00" nor "10", i.e., when the LLQSs are mixtures of 0s and 1s. It is clear that by recording the LLQSs separately from the quadtree nodes, the bitplane bitmap cells do not need to be represented as the quadrant signatures in the leaf nodes of a BQ-Tree with values of either "00" or "10" and thus the

aforementioned redundancy is avoided. It can also be seen that, while the BQ-Tree data structure does not explicitly store the positions of the compacted quadrant signatures in its leaf tree nodes, both encoding and decoding algorithms can utilize the implicit correspondence when the tree node array and the LLQS array are processed in a streamline manner.

Our BQ-Tree design allows arbitrary last level quadrant sizes of powers of 2. For a last level quadrant of size 2*2, we will need to combine two of such quadrant signatures (each is 4 bits) to form a byte. For a last level quadrant of size 4*4, it is natural to use a short integer (16 bits) to represent the quadrant signatures. Fig. 2 shows the processes of compacting LLQSs using both 2*2 and 4*4 quadrant sizes. The bitmap shown in the left part of Fig. 2 represents a bitplane of an 8*8 raster. Using a 2*2 last level quadrant size, a BQ-Tree with two levels (as shown in the top of Fig. 2) is generated. Breadth-first traversal of the tree generates the tree node array as 01100100 1010010 00001001. Note that the second and the fourth level-2 tree nodes are skipped as the corresponding child node signatures in the root node are "10" and "00", respectively. Combing the two last level quadrants that are the mixtures of 0s and 1s (shaded in the top-left part of Fig. 2) generates the first byte of the LLQS array. Note that the XXXX (a half-byte) at the end of the array is a filler to make a whole byte. On the other hand, using the 4*4 last level quadrant size generates a BQ-Tree with only one level. Similar to using 2*2 quadrant size, quadrants with mixtures of 1s and 0s are sequenced on the LLQS array (short integer data type in this case). It is clear that using larger last level quadrant sizes will reduce BQ-tree depths and numbers of tree nodes at the cost of increasing the LLQS array volume.

4 PCL BASED GPGPU DECODING

In this study, we aim at making full use of GPGPUs' massive parallel computing capabilities to speed up decoding large-scale geospatial raster from encoded BQ-Trees. CUDA has two levels of parallelism: block level and thread level [8]. Assigning a chunk to a CUDA computing block is very similar to the coarse-grained parallelization on CPUs which is relatively straightforward. However, assigning hierarchically encoded BQ-trees of a raster chunk to a group of flatly organized GPGPU threads within a computing block requires a more careful design. Among various designs and implementations that we have tried, an approach called Process Collectively and Loop (PCL) achieved the best results and will be presented in details next.

In PCL, all the threads assigned to a computing block are bundled together to process a quadrant of matrices in a BQ-Tree pyramid during decoding (Fig. 3). This collective process is looped over all the quadrants and all levels of the pyramid. The PCL approach requires the starting positions in both the tree node array and the LLQS array in order to make the threads assigned to the GPGPU computing blocks work in parallel. However, keeping the positions for tens of thousands of GPU threads is impractical since the storage overhead would be overwhelming. Our solution is to calculate the positions for all threads on the fly. While the idea is straightforward, unfortunately, synchronizing GPGPU computing blocks, which is required to calculate the positions for quadrants across computing blocks, is very costly and inflexible in the current generations of GPGPUs. The PCL approach adopts a strategy that requires pre-generating the positions for every T_n elements that are processed in a computing block while computes the positions on the fly for all the T_n threads within a computing block. We believe the hybrid strategy provides a good tradeoff and the efficiency has been verified by the experiments. Given a raster chunk $C=M*M=2^m*2^m$, clearly the storage overheads for the starting positions for the tree node array and the LLQS array can be calculated as $S_n=B*(1+4+\ldots+4^{m-t-1}) = B*(4^{m-t}-1)/3$ and $S_l=B*4^m/T_n$, respectively. We note that both S_n and S_l decrease as T_n increases ($T_n=2^t*2^t$). We next present the details of the on-the-fly calculation of the positions in a GPGPU computing block.

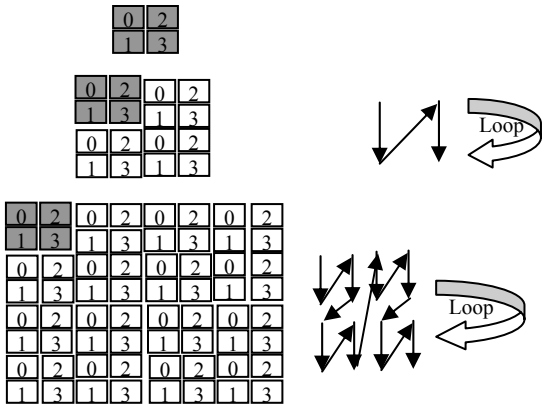

Fig. 3 Illustration of the PCL Parallelization Scheme

Assuming that the starting position of a data segment with T_n data elements (tree nodes or LLQSs) in the whole array is p, the task is to compute the positions for the T_n threads from where the threads can fetch data from global memory and perform decoding independently. In our implementation, first, the numbers of child nodes (or non-uniform last level quadrants) are counted by examining the parent nodes and counting the numbers of quadrants with "01" signature. These values are then written to an array of size $2*T_n$ in the computing block's fast shared memory. The position offsets of all the threads relative to the first thread (whose position is known as p) can then be calculated through a fast parallel scan process in the shared memory with a time complexity in the order of $O(\log T_n)$ [9]. After the starting positions of the threads are computed, all threads can work independently to decode quadtree nodes by retrieving the node signatures from the compressed quadtree byte stream for quadrants with "01" signatures. The decoded quadtree nodes are then written to global memory for next level decoding. We note that the PCL scheme does not incur additional accesses to global memory as the on-the-fly calculation of positions is all done in fast shared memory. Experiments have shown that the calculation cost is negligible on GPUs. In addition, the PCL scheme assures that global memory is always accessed continuously by the threads of the computing blocks.

In our implementation, a separate kernel is launched to combine the decoded bitplane bitmaps and restore the original raster. Although this requires additional global memory accesses, it significantly reduces register consumptions which subsequently eliminate register spilling to global memory. The implementation of the combination kernel is quite simple as the decoded bitplane bitmaps are now regularly shaped matrices which are excellent for coalesced global memory accesses. Since the raster dataset in our experiments is the 16-bits short integer type and there are 16 bitplane bitmaps represented as 8-bits chars, a working array of 8 short integers is used to hold intermediate results. Each thread reads a byte from 16 bitmaps, converts them to 8 short integers and writes the short integers to global memory. The implementation is easy to be modified for 8 and 32 bits rasters.

5 PERFORMANCE STUDIES

We use a real NASA MODIS (Moderate Resolution Imaging Spectroradiometer) raster dataset obtained from the Global Land Cover Facility (GLCF) website [10]. The specific dataset we use is band1 of the North America 2003097 imagery. The dataset has a spatial resolution of 500 meters and 22,658*15,586 cells sampled at 16-bits. The original data volume is 706,295,176 bytes. A downscaled image is shown in Fig. 4 to help understand the dataset better. The image clearly shows that a signficant portion of the dataset is covered by oceans whose raster cell values are mostly NO_DATA or some other special masks. Among the cells with valid values (1-16000), 75.8% cells have values that are less than 4096, i.e., the first 4 bitplane bitmaps (most significant bit first) are mostly 0s. As such, we can expect significant compression when the dataset is encoded by BQ-Trees.

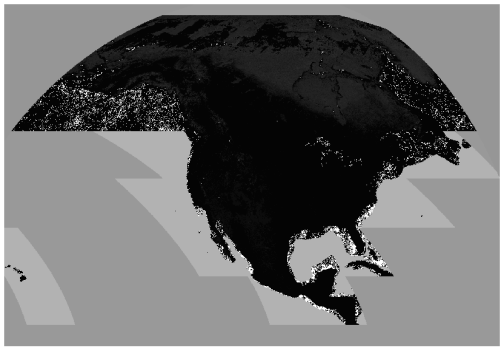

Fig. 4 Grayscale Image of the Experiment Dataset

Our experiments are performed on a two-node SGI Octane III machine equipped with dual Intel Xeon E5520 quadcore CPUs (hyper-threading enabled), 48 GB memory and two Nvidia C2050 GPU cards. Only one node and one GPU card are used for our experiments. Our primary measurement in this study is the wall-clock running times measured in milliseconds. We have tested all the CPU and GPGPU implementations using two chunk sizes: 1024*1024 and 4096*4096. Using small chunk sizes can certainly reduce the volumes of padded data when the original rasters are not the exact multiplications of the chunk sizes and can potentially reduce workloads and improve performance. On the other hand, using large chunk sizes can reduce hardware scheduling overheads which could be beneficial in certain cases.

We have chosen to use 256 threads per computing block, i.e., T_n=256, for all the GPGPU experiments to achieve an optimum GPGPU hardware occupancy after considering constraints related to registers and shared memory. The PCL implementation achieves a performance of 190 milliseconds when the chunk size is set to 1024*1024 and last level quadrant size is set to 2*2. On the other hand, the best CPU result running on a dual quad-core machine with 8 cores and 16 threads is 1095 milliseconds after turning on the O3 optimization. The best single thread CPU result is 7005 milliseconds also with O3. As such, our GPGPU result has achieved a 5.8X speedup (1095/190) when compared with the 16-thread CPU implementation and a 36.9X speedup (7005/190) when compared with a single thread CPU implementation.

6 SUMMARY AND CONCLUSIONS

In this study, we have developed the BQ-Tree spatial data structure to code large-scale raster geospatial data. We have developed a fine-grained parallelization scheme, namely PCL (Process Collectively and Loop), and an implementation by using both system and application level optimization strategies. Experiments show that PCL is capable of decoding a BQ-Tree encoded 16-bits NASA MODIS image with 22,658*15,586 cells in 190 milliseconds, i.e., 1.86 billion cells per second, on an Nvidia C2050 GPU card. The performance achieves nearly a 6X speedup than the best dual quadcore CPU implementation using 16 threads (8 cores) and achieves nearly a 37X speedup than a single thread CPU implementation.

For future work, first of all, we would like to follow the multi-component bitmap indexing framework introduced in [11] to code geospatial rasters that can facilitate exact, range and interval queries, in addition to bitplane level decoding (the focus of this study). By integrating these two components, we plan to develop a working prototype system with performance accelerated by multicore CPUs and GPGPUs to visually explore large-scale raster geospatial data and support global environmental studies. Second, while the GPGPU-based design and implementation are compared against CPU-based ones in this study, from a practical perspective, it is more useful to integrate multicore CPU and GPGPU implementations to achieve the best performance. We would like to develop cost models and efficient scheduling algorithms for this purpose. Third, while SciDB [5] currently only considers parallelization of managing multidimensional array data on shared-nothing clusters, we would like to reuse SciDB and FastBit codebases to develop a plugin module to facilitate managing high-resolution, time-evolving and multi-variant raster geospatial data on commodity desktop computers equipped with multicore CPUs and GPGPUs.

7 REFERENCES

1. Gaede, V. and O. Gunther (1998). Multidimensional access methods. ACM Computing Surveys **30**(2): 170-231.
2. Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann.
3. Kothuri, R. K. V., S. Ravada, et al. (2002). Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. ACM SIGMOD conference, 546-557.
4. Fang, Y., M. Friedman, et al. (2008). Spatial indexing in Microsoft SQL server 2008. ACM SIGMOD conference, 1207-1216.
5. SciDB. http://www.scidb.org/.
6. FastBit. https://sdm.lbl.gov/fastbit/.
7. Wu, K. S., E. J. Otoo, et al. (2006). Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems **31**(1): 1-38.
8. Nvidia. Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_home_new.html
9. Satish, N., M. Harris, et al. (2009). Designing efficient sorting algorithms for manycore GPUs. IEEE Symposium on Parallel\&Distributed Processing, 1-10.
10. Global Land Cover Facility (GLCF) MODIS 500m North America Dataset, ftp://ftp.glcf.umiacs.umd.edu/modis/
11. Wu, K., A. Shoshani, et al. (2010). Analyses of multi-level and multi-component compressed bitmap indexes. ACM Trans. Database Syst. **35**(1): 1-52.