A Quadtree-Based Lightweight Data Compression Approach to Processing Large-Scale Geospatial Rasters

Jianting Zhang[1,2] and Simin You[2]
1 Department of Computer Science, City College of New York, New York, NY, 10031
2 Department of Computer Science, CUNY Graduate Center, New York, NY, 10006
Correspondent author email: jzhang@cs.ccny.cuny.edu

Abstract

Huge amounts of geospatial rasters, such as remotely sensed imagery and environmental modeling output, are being generated with increasingly finer spatial, temporal, spectral and thematic resolutions. Given that CPUs on modern computer systems are three orders of magnitude faster than disk I/O speed and two orders of magnitude faster than network bandwidth, it becomes more and more beneficial to allocate computing power for real time compression and decompression to reduce I/O times in streaming large-scale geospatial rasters among CPU memory, disks and distributed computing nodes and file systems. In this study, we aim at developing a lightweight lossless data compression technique that balances the performance between compression and decompression for large-scale geospatial rasters. Our Bitplane bitmap Quadtree (or BQ-Tree) based technique encodes the bitmaps of raster bitplanes as compact quadtrees which can compress and index rasters simultaneously. The technique is simple by design and lightweight by implementations. Apart from computing Z-order codes for cache efficiency, only bit level operations are required. Extensive experiments using 36 rasters of the NASA Shuttle Range Topography Mission (SRTM) 30 meter resolution elevation data with 20 billion raster cells have shown that our BQ-Tree technique is more than 4X faster for compression and 36% faster for decompression than zlib using a single CPU core while achieving very similar compression ratios. Our technique further has achieved 10-13X speedups for compression and 4X speedups for decompression using 16 CPU cores. On the experiment machine equipped with dual Intel Xeon 8-core E5-2650V2 CPUs, our technique is able to compress the SRTM raster dataset in about 35 seconds and decompress back in 29 seconds. The performance compares favorably with the best known technique with respect to both compression and decompression throughputs. We have made the source code package and a subset of the SRTM raster publically available to facilitate validations and cross-comparisons.

1. Introduction

Advancements in remote sensing technologies and environmental modeling have generated huge amount of large-scale geo-referenced raster data (or geospatial rasters for short). While multi-core CPUs are capable of performing 10-100 billions of floating point operations per second (Gflops), the disk I/O speed is typically limited to 50-100 megabytes per second (MB/s) and the network bandwidth is limited to 0.1-1 gigabytes per second (GB/s). There are three orders of magnitude of difference between CPU and disk I/O speeds and two orders of magnitude of difference between CPU speed and network bandwidth (Hennessy and Patterson, 2012). Disk I/O and network bandwidth are among the most significant bottlenecks in processing large-scale geospatial rasters. For example, when processing the NASA Shuttle Range Topography Mission (SRTM) 30 meter resolution Digital Elevation Model (DEM) data[1], the 20 billion raster cells in the Continental United Sates (CONUS) region amounts to 38 GB data in uncompressed format and it takes 400+ seconds to load the data to memory on a typical personal workstation. While many geospatial processing tasks are traditionally computing bound which makes them less sensitive to low disk I/O speed, the fast growing computing power on parallel hardware, such as multi-core CPUs and many-core Graphics Processing Units (GPUs), has made disk I/O and network bandwidth outstanding issues in many applications that involve processing large-scale geospatial rasters.

Data compression is a popular approach to reducing data volumes and hence lowering disk I/O and network data transfer times. While several lossy data compression tasks have demonstrated excellent compression ratios, lossless data compression techniques are still among the most popular ones in geospatial processing due to data quality and accuracy concerns. The Linux tool gzip[2] and its underlying zlib[3] library package might be among the most popular tools for general purpose lossless compression, including geospatial rasters, with demonstrated excellent performance. However, the library package is fairly complicated in order to support possible optimizations of implementations of the underlying LZ77 algorithm (Salomon 2006) which is already considerably sophisticated. This makes zlib heavyweight, less flexible to new hardware architectures, and difficult to incorporate domain knowledge for better performance. In addition, the LZ77 algorithm is asymmetric by design and it is much more expensive to compress than decompression. While this is a good choice in many applications that involve more decompression than compression, it is not well suitable for quite some geospatial processing applications that symmetric algorithms and implementations are more preferable. For example, many intermediate results of environmental modeling are only compressed and decompressed once for the sole purpose of reducing disk or network traffic in distributed computing. The frequencies of compression and decompression are roughly the same and very often data compression and decompression are interleaved with other types of computation. As such, comparing with sophisticated and asymmetric approaches (such as zlib), symmetric and lightweight techniques are more preferable.

In this study, we aim at developing a new lightweight data compression technique for large-scale geospatial rasters, with respect to both algorithmic complexity and computing time, to effectively reduce data communication time due to disk and/or network I/Os. The proposed technique is based on our Bitplane bitmap Quadtree (BQ-Tree for short) structure that was originally developed for indexing geospatial rasters on GPUs (Zhang et al 2011). The basic idea of BQ-Tree is to represent all the bits of all raster cells in a bit position as a bitmap and then use a quadtree structure to code the bitmap efficiently. For a K-bit raster, there will be K bitplane bitmaps and the resulting K BQ-Trees will be concatenated as the compressed data. Similar to many data compression techniques, large rasters can be segmented in to chunks (e.g., 1024*1024 cells in a chunk) to facilitate parallelization and/or bound computation overhead.

[1] http://www2.jpl.nasa.gov/srtm/
[2] http://www.gzip.org/
[3] http://www.zlib.net/

Comparing with zlib, our technique is conceptually simple and can be implemented in a few hundreds of lines of code (including both compression and decompression algorithms) and embedded in various applications. Although further optimizations are quite possible, comparing with zlib, experiments on our current serial and parallel implementations using NASA SRTM 30 meter elevation data with 20 billion 16-bit raster cells (38 GB) have demonstrated favorable results on a dual Intel Xeon 8-core E5-2650V2 machine with 16 CPU cores running at 2.6 GHZ. First of all, our technique is significantly faster (4.1X) than zlib for compression and faster (36%) for decompression when using a single CPU core in the serial implementation. Using the same chunk size of 1024*1024 but using all the 16 CPU cores (parallel implementation), our technique is 6.5X faster than zlib for compression, although it is 1.7X slower for decompression, possible due to memory bandwidth contention on the multi-core CPUs when all cores are utilized. Second, the compression ratios are comparable with zlib. Using a chunk size of 1024*1024, the compressed size of the SRTM data is about 7.2 GB using both zlib and our technique. A very similar compression ratio is obtained using a chunk size of 4096*4096. Interesting, by applying zlib on top of our BQ-Tree compressed raster tiles, the compressed size is further reduced to about 5.5 GB. Correspondingly, the compression ratio is improved from 0.1891 to 0.1458. This suggests that our BQ-Tree technique can be further optimized to achieve better compression ratios than zlib when the tradeoff between computing workload and compression ratio can be justified.

Finally, comparing with the best published lossless compression technique for scientific data (ISOBAR, Schendel et. al. 2012a), our technique is comparable or even has a higher performance with respect to both compression throughput (~125 MB/s) and decompression throughput (~400 MB/s) using a single CPU core. We have further achieved 10-13X speedups for compression and about 4X speedup for decompression using 16 CPU cores. Overall, our technique can achieve about 1.7 GB/s throughput rates for both compression and decompression using the 16 CPU cores. We expect it to achieve comparable performance on similar multi-core CPU machine. The overall high-performance with respect to compression and decompression can be attractive to many practical geospatial processing applications. To facilitate repeating our experiments and encourage adoptions and comparisons, we have made the source code package and a sample raster (41400*18000, 16-bit) at our website[4].

The rest of the paper is arranged as follows. Section 2 introduces background, motivation and related work. Section 3 presents the BQ-Tree designs and implementations on multi-core CPUs. Section 4 provides experiment results on NASA SRTM 30 meter elevation data and compares our technique with alternative ones. Finally, Section 5 concludes the paper and discusses future work directions.

2. Background, Motivation and Related Work

Geospatial processing has been undergoing a paradigm shift in the past few years. Advancements in remote sensing technology and instrumentation have generated huge amounts of remotely sensed imagery from air- and space-borne sensors. In recent years, numerous remote sensing platforms for Earth observation with increasing spatial, temporal and spectral resolutions have been deployed by NASA, NOAA and the private

[4] http://www-cs.ccny.cuny.edu/~jzhang/bqtree_coding.htm

sector. The next generation geostationary weather satellite GOES-R serials[5] (whose first launch is scheduled in 2016) will improve the current generation weather satellites by 3, 4 and 5 times with respect to spectral, spatial and temporal resolutions (Schmit et al 2009). With a temporal resolution of 5 minutes, GOES-R satellites generate 288 global coverages everyday for each of its 16 bands. At a spatial resolution of 2km, each coverage and band combination has 360*60 cells in width and 180*60 cells in height, i.e., nearly a quarter of a billion cells. While 30-meter resolution Landsat TM satellite imagery is already freely available over the Internet from USGS[6], sub-meter resolution satellite imagery is becoming increasing available, with a global coverage in a few days. Numerous environmental models, such as Weather Research and Forecast (WRF[7]), have generated even larger volumes of geo-referenced raster model output data with different combinations of parameters, in addition to increasing spatial and temporal resolutions. For example, the recent simulation of Superstorm Sandy[8] on National Science Foundation (NSF) Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA) has a spatial resolution of 500 meters and a 2-second time step. Running at 9120*9216*48 three-dimensional grid (approximately 4 billion cells), a single output file is as large as 264 GB (Peter et al 2013).

The large volumes of geospatial raster data have imposed a significant challenge on data management. Despite storing large amounts of data is getting cheaper and cheaper due to decreasing cost of magnetic disks, moving such data through various processing pipelines are getting more and more expensive when compared with floating point computation. While many geospatial processing tasks are relatively insensitive to low disk I/O speeds and network bandwidths in the past due to low CPU processing power, the fast growing computing power on parallel hardware, such as multi-core CPUs and many-core GPUs has made disk I/O and network bandwidth outstanding issues in many applications that involve large-scale geospatial rasters. It is thus desirable to allocate more computing power to data compression and reduce data volumes that need to go through disk I/O controllers and network interfaces to improve overall system performance. As a well established research discipline, many lossy and lossless data compression techniques have been developed in the past few decades, especially for text and image data (Salomon 2006). However, many of these data compression techniques are optimized for compression ratios and/or data transmission over noisy data communication channels (e.g., wireless). The sophisticated designs not only require complex implementations but also make them difficult to parallelize. For example, to the best of our knowledge, despite several efforts (e.g., Ozsoy et al 2014), efficient implementations of the LZ77 algorithm on GPUs are still in early stages.

Although it seems to be straightforward to simply apply image compression techniques to geospatial rasters to achieve better compression ratios than the generic tools such as zlib, there are several concerns. First of all, many image compression techniques are lossy in nature. While they may perform well in terms of both compression ratios and computing overhead when operated under the default lossy compression modes, the performance can be significantly poorer if lossless compression is required even if the

[5] http://www.goes-r.gov/
[6] https://lta.cr.usgs.gov/SRTM2
[7] http://www.wrf-model.org
[8] http://en.wikipedia.org/wiki/Hurricane_Sandy

techniques may allow lossless compression as an option. Second, many geospatial rasters use special values (e.g., -1 or 65535) to represent outliers (e.g., no values, masked cells). These outliers may have significantly different values than their neighbors and may generate large coefficients for many transformation based techniques, such as wavelets and delta coding (Salomon 2006). Third, while it is not mandatory to integrate indexing with compression in all cases, from an application perspective, it is more beneficial to be able to use data structures embedded in compressed data to facilitate indexing, querying and other types of processing which is becoming increasingly popular in practical applications (e.g., Jenkins et al 2013, Gong et al 2012). Scientific applications that involve multi-dimensional arrays (including geospatial rasters) that require lossless data compression rarely use image based data compression techniques, despite the fact that image and raster data types are conceptually similar. The first two concerns might explain the observation to a certain extent. On the other hand, the lossless compression techniques for generic data, such as LZ77/zlib, while still popular in practice, as discussed earlier, suffers from complexity and long encoding time.

Our BQ-Tree based compression technique is designed to fill the gaps. First of all, it is a conceptually symmetric lossless compression technique by design and can be easily implemented across multiple parallel hardware platforms. In addition to serial and multi-core CPU based parallel implementations (using simple OpenMP[9] directives) for both compression and decompression as reported in this study, we have also developed a GPU-based decoding (decompression) implementation as reported in our previous work (Zhang et al 2011). We are in the process of developing implementations that can utilize the Vector Processing Units (VPUs) that come with CPUs on modern hardware (Hennessy and Patterson, 2012) for both compression and decompression as well as a GPU-based compression module for higher performance. The new implementations and more comprehensive comparisons will be reported in our future work. On the other hand, we note that pure CPU implementations (without using VPUs or GPUs) make it possible to apply our technique on older generations of CPUs, which can be desirable in some cases.

Second, our technique allows simple implementations while achieving impressive performance. The building blocks of our modestly optimized implementations of both compression and decompress algorithms amount to roughly 300 lines of code which can be easily integrated in many applications, including the possibility for further domain-specific optimizations. The bitplane level data transformation makes sliced bitmaps more homogeneous and suitable for quadtree based compression and decompression. While the transformation resembles wavelet based data compressions in a sense, it is much simpler with respect to required arithmetic operations. Compared with wavelet transformations and delta encoding that typically perform poorly when there are outstanding entries that are significantly different from neighbors, our technique is less sensitive to such outliers (including NODATA). As an example, while NODATA requires special handling when JPEG2000 is used to encode remote sensing images as proposed in (Gonzalez-Conejero et al 2010), such special handling is not necessary in our technique.

Third, different from some generic compression techniques that are primarily optimized for compression (zlib in particular), our BQ-Tree based technique originated from raster indexing and query processing and naturally supports several types of queries

[9] http://openmp.org/wp/

and analytics efficiently (see discussions in Zhang et al 2011). Although indexing and query processing are traditionally important research topics in database communities, they are becoming increasingly important in high-performance computing research on supercomputers (Jenkins et al 2013, Lakshminarasimhan et al 2013a, Gong et al 2012, Chou et al 2011, Lin et al 2009, Gosink et al 2006). Our tree-based technique is significantly different from existing techniques that are based on flat indexing structures, such as bitmaps (Chou et al 2011, Lin et al 2009, Gosink et al 2006) and inverted lists (Jenkins et al 2013, Lakshminarasimhan et al 2013a, Gong et al 2012). Different from these techniques that store compressed data and indices separately, the resulting tree structure in our technique is part of the compressed data. In addition, when only certain values or ranges are interesting to users, the tree structures can be used to locate the right chunks of data directly without needing to go through all the grid cells. This feature is particularly useful when only such interesting values are required in subsequent processing, as large portions can be either discarded or simply streamed to external storage without decompression.

The most relevant work to ours is the In-Situ Orthogonal Byte Aggregate Reduction (ISOBAR) lossless compression technique for scientific data due to Schendel et al (2012a). The technique performs favorably over zlib in terms of both throughput and compression ratio. ISOBAR and BQ-Tree share the similarity of searching for compressible blocks in a top-down manner. However, ISOBAR searches for arbitrarily shaped flat rectangular blocks at byte level while BQ-Tree searches for homogenous quadrants at bit level in a hierarchical manner. While it is possible to support query processing based on the derived block level metadata in ISOBAR by scanning all the blocks, the hierarchical tree structure derived by the BQ-Tree approach could be more efficient in query processing. Our BQ-Tree technique has comparable performance on a single CPU core based on the results reported in (Schendel et al 2012a). However, no parallel results have been reported in (Schendel et al 2012a) and the ISOBAR's scalability on multi-core CPUs is thus unclear. As detailed in Section 4, the BQ-Tree technique achieves good scalability on a 16-core machine for compression (10X-13X). For decompression on parallel hardware with large a number of processing cores, memory bandwidth may become a bottleneck as bitmaps of multiple bitplanes need to be combined which significantly lowers the achievable multi-core speedup (4X for 16 cores). On the other hand, given that the decompression throughput has reached 250-1600 MB/s according to (Schendel et al 2012a), it is likely that memory bandwidth will also be a problem for ISOBAR on multi-core CPUs. As ISOBAR and our technique have similar goals but are quite different in designs, it is interesting to perform a comprehensive comparison with respect to algorithmic complexity, memory bandwidth requirement, scalability, compression ratio and end-to-end runtime. We leave this for future work when ISOBAR code becomes publically available.

Compared with previous works on lossless compressions of multi-dimensional array data, there are more works on lossy compressions, especially for images and videos (Salomon 2006). Various transformations, including cosine, Fourier, wavelet can be applied to increase homogeneity for better compression. Delta coding is especially effective for sorted data (Lemire1 and Boytsov, in press). The In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) approach proposed in (Lakshminarasimhan et al 2013b) integrates sorting and spline-based predicative coding for scientific data with demonstrated good performance. Elevation data has also been extensively used in computer graphics for visualization purposes and several lossy compression techniques have been proposed (Lindstrom and Cohen 2009, Durdevic et al 2013). However, when elevation data is used for spatial analysis and other scientific inquiries, lossless compressions are typically preferred, despite some lossy compression techniques do have the capabilities to guarantee error bounds. Similar to elevation data, many environmental data in the form of geospatial rasters, either captured from remote or in-situ sensors or derived from environmental models, have different requirements with respect to compression: lossy compression for visualization and lossless compression for analysis. Ideally, compressed data can have certain structures to support efficient queries, based on either metadata or indices. Since it is typically expensive to maintain multiple compressed copies of large datasets with different compression formats, losslessly compressed data that tightly integrates indexing structures is preferred. However, to the best of our knowledge, most of the existing techniques, including ISOBAR, are solely developed for data compression without supporting query processing. On the other hand, techniques such as FastBit[10] that are specifically designed for indexing multi-dimensional scientific data (Wu et al 2006, Wu et al 2010) and has been demonstrated to be efficient in query processing (Chou et al 2011, Lin et al 2009, Gosink et al 2006). However, although the index sizes are significantly better than B+-Trees (Wu et al 2004), they may result in large (up to 200%) index file sizes (Gong et al 2012, Jenkins et al 2013, Lakshminarasimhan et al 2013) and actually increase disk I/O overheads.

Orthogonal to data compression approaches to reducing disk and network I/Os, alternative approaches to speed up data intensive computation include adapting data layouts based on locality (Gong et al 2012, Liu et al 2013a, Gong et al 2013), using multi-level storage (Soroush et al 2011) or multi-resolution representation (Tian et al 2013), embedding analytics within I/Os (Kanov et al 2012) and using parallel disk I/Os through fast communication networks (Lofstead et al 2011, Liu et al 2013b, Zheng et al 2013). While some of these techniques have been exploited in conjunction with data compressions, we believe our BQ-Tree based technique that tightly combines compression with indexing has the potential to be integrated with these techniques for better performance in large-scale geospatial processing in both personal and cluster computing environments.

3 Bitplane Quadtree

3.1 The BQ-Tree Data structure

A K-bit raster is first split into K bitplanes and each bitplane becomes a bitmap. Given a bitplane bitmap of a raster R of size N*N (without losing generality, assuming $N=2^n$), as illustrated in Fig.1, 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

[10] https://sdm.lbl.gov/fastbit/

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.

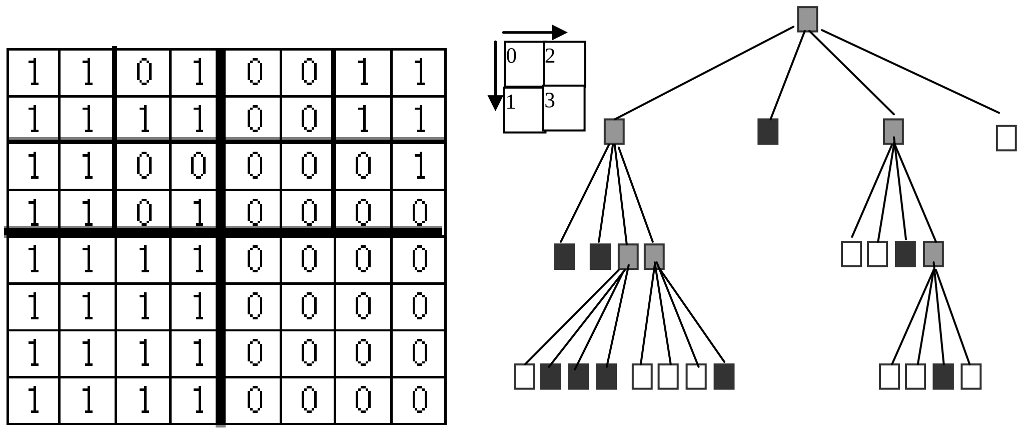

Fig. 1 Illustration of quadtree representation of a bitplane bitmap

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 (Morton 1966) as shown in 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 a 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.

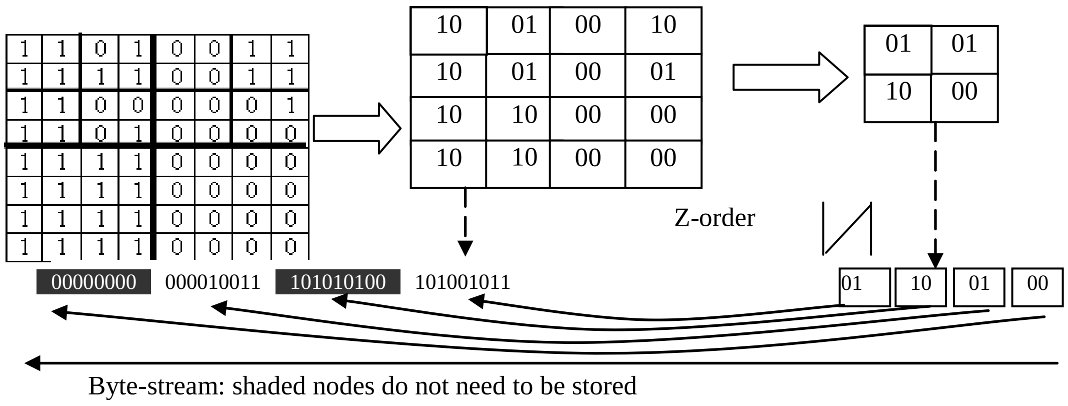

Fig. 2 Streaming BQ-Tree Nodes

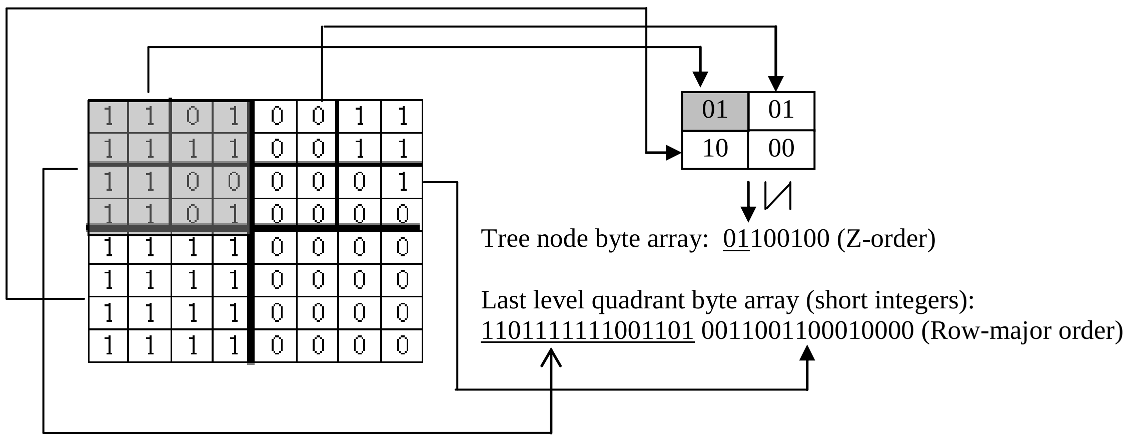
Tree node byte array: 01100100 (Z-order)
Last level quadrant byte array (short integers):
1101111111001101 0011001100010000 (Row-major order)

Fig. 3 Generating LLQS Array Using a 4*4 Quadrant Size

In this study, we use a 4*4 last level quadrant size and an example to show the processes of compacting LLQSs in Fig. 3. The bitmap shown in the left part of Fig.3 represents a bitplane of an 8*8 raster. The resulting quad-tree has only one node and its signature is 01100100. This is because the signature of the shaded quadrant is 01 (mixture of 0s and 1s) and the signatures of the rest of the three quadrants are 10 (all 1s), 01 (mixture of 0s and 1s) and 00 (all 0s), respectively. The quadrant signatures are concatenated to form the quadtree node for the 8*8 bitmap. Since the first and the third quadrants are mixed of 0s and 1s, their last-level quadrant signatures need to be streamed to the LLQS array, which results in the concatenation of two 16-bit short integers, i.e., $(1101111111001101)_2$ and $(0011001100010000)_2$. Note that the bits of last level quadrants follow row-major order instead of Z-order to minimize Z-order calculation as our experiments have shown that Z-order calculation can be expensive.

While our primary focus in this study is compression and decompression, the BQ-Tree structures can be used to support efficient query processing in a way similar to FastBit (Wu et al 2006, Wu et al 2010). Indeed, a BQ-Tree can be considered as a hierarchically compressed multi-dimensional bitmap and thus many bitmap related query processing techniques can be applied, including exact value queries and range based queries. Query results can also be compressed using the same technique and stored in memory or disk efficiently. This can be very useful to identify Regions of Interests (ROIs) that satisfy multiple query criteria (conjunctive, disjunctive or combinations) on multiple rasters which is increasingly important in visual explorations of large-scale rasters. The quadtree structure of our technique could be more efficient in labeling ROIs than flatly structured bitmaps as used in (Sinha et al 2009).

3.2 BQ-Tree based Compression

While the BQ-Tree structure was primarily designed for indexing large-scale geospatial raster and was not optimized for data compression, as nearby cells of geospatial rasters typically have similar values and their higher bits are often the same, it can be naturally used for lightweight geospatial raster compression.

```
Step 1: for each of the $2^{m-q}$ *$2^{m-q}$ last level quadrants
  1.1 Gather the raster cells in the quadrant and calculate the Z-order code of the quadrant
  1.2 For each of the K bits
    1.2.1 Generate the last level quadrant signatures and write them to LA
    1.2.2 Derive the child node signatures for four neighboring quadrants and form a leaf
          node; output the leaf node to the last level (level m-q) matrix of PA
Step 2: For each of the m-q-1 levels of PA, loop bottom-up along the pyramid and do the
        following
  2.1 For each of the elements of the matrix at the level l
    2.1.1 For each of the K bits do the following:
      2.1.1.1 Examine the four child nodes at the level l+1 matrix and generate the signatures
              for the four quadrants of the node using the following rules: 0x00->"00", 0xaa->"10", all
              others ->"01".
      2.1.1.2 Concatenate the four 2-bits signatures and write the node value to the level l
              matrix
Step 3: For all the elements in PA, output bytes that are not 0x00 or 0xaa
Step 4 For all the elements in LA, output short integer values (2 bytes) that are not
       0x0000 or 0xFFFF (for last level quadrant size of 4*4).
```

Fig. 4 BQ-Tree based Data Compression Algorithm

The inputs for the compression algorithm are raster (or raster chunk) R, raster/chunk size C ($C=2^m*2^m$), last level quadrant size S ($S=2^q*2^q$) and K, which is the number of bits of R. The outputs of the algorithm include the compacted BQ-Tree node array CN and the compacted LLQS array CL. The initialization includes allocating a pyramid array for all bitplanes (PA) and allocating a LLQS array (LA). Clearly the size of PA should be $K*(1+4+16+\ldots+2^{m-q-1}*2^{m-q-1})=K*(4^{m-q}-1)/3$ and the size of LA should be $K*2^{m-q}*2^{m-q}$. The encoding algorithm is given in Fig. 4 which is straightforward to follow by using the example provided in Fig. 3. First, a whole raster (or a raster chunk) is divided into quadrants based on the last level quadrant size. Both the signatures of the last level quadrants and the corresponding leaf nodes are then generated (Step 1). Second, for each bitplane, a pyramid is generated bottom-up by combining the child node signatures into the parent node signatures (Step 2). Third, starting from the root of the BQ-tree for each bitplane, all the nodes in the matrix correspond to a pyramid level are examined by following the Z-order. The pyramid is then compacted into a byte array for each bitplane by skipping 0x00 (all 0s) and 0xaa (all 1s) bytes (Step 3). Finally, the signatures of the last level quadrants are also compacted into a short integer stream by skipping signatures that are considered to be uniform, i.e., those correspond to “00” or “10” values in any of the four quadrants of the leaf nodes of a BQ-Tree (Step 4).

3.3 BQ-Tree based Decompression

The decoding process is the reverse of the encoding process. A detailed procedure similar to Fig. 4 can be easily constructed and is skipped in the interest of space. Instead, we next present an overview of the steps of the decompression algorithm and discuss several implementation considerations. Starting from the root of a BQ-Tree, the pyramid PA is reconstructed level by level as follows. Each quadtree node is scanned and the signatures of the four child nodes are extracted and examined. Values of 0x00 and 0xaa will be used to update the corresponding matrix elements in the next level (i.e., child nodes in the pyramid layout) if the child node signatures are “00” or “10”, respectively. Otherwise, a byte value is retrieved from the compacted BQ-Tree byte stream and used to update the corresponding matrix elements in the next level. After the pyramid is reconstructed, the elements of the last level matrix of the pyramid (correspond to the leaf nodes of the BQ-Tree) are then combined with the LLQS array to reconstruct the original bitplane bitmap by setting the LLQSs with either all 0s and all 1s if the quadrant signatures in the leaf nodes are 00 or 10, respectively; or with the values in the LLQS array if the quadrant signatures in the leaf nodes are 01. Finally, the reconstructed bitplane bitmaps are combined to reconstruct the raster cell values through bitplane level composition. To set the i^{th} bit of raster cell value v decoded from the i^{th} BQ-Tree of a raster chunk, the following bitwise operation can be applied: v|=(b<<i) where b is the bit value of the i^{th} bitplane of the raster cell.

3.4 Discussions on Key Implementation Issues

It is clear that algorithms for both compression and decompression are conceptually simple as only bit-level operations are involved, except Z-order related calculation which may require significant amount of arithmetic operations if it is implemented in a naive way. Fortunately, by using the efficient design proposed in (Raman and Wise 2008), the conversion between (row, column) pairs and Z-order codes is mostly reduced to table lookups which are highly efficient, provided that the lookup tables can be accommodated in on-chip CPU caches. Since most reasonably up-to-date CPUs have at least 32KB L1 cache, despite that the cost of Z-order related computation is still significant, it does not dominate based on our experiments. Nevertheless, it is still important to reduce the numbers of Z-order code computation as much as possible. By using row-major order instead of Z-order for all the raster cells in a last level quadrant, we have reduced the number of Z-order computation to 1/16 by using a 4*4 last level quadrant size. While it is interesting to use larger last level quadrant sizes to further reduce Z-order related computation overhead, we note that row-major order performs poorly when the raster cells in a last level quadrant cannot fit in a cache line, which is a well known issue in accessing multi-dimensional arrays (Hennessy and Patterson, 2012). Using a 4*4 last level quadrant size, 32 bytes are needed to store all the 16 raster cells (2 bytes each) in a last level quadrant. This is smaller than L1 cache line size in most CPUs and works fine. Although some CPUs may have 128-byte L1 cache line size and thus support the 8*8 LLQS size naturally, as discussed in Section 3.1, using a larger last level quadrant size may potentially decrease compression ratio in non-homogenous regions in a raster, although it may reduce runtimes of both compression and decompression.

Different from compression that only a single Z-order computation is required for a last level quadrant, if we combine the K decompressed signatures of a last level quadrant directly to the output array, we will need K individual Z-order computations, each for a bitplane. An optimization we have adopted is to use a temporal array to hold all the bitplane-combined last-level quadrants in the order of quadtree nodes, i.e., Z-order. Note that raster cells within a last-level quadrant are still in row-major order. We compute Z-order values only when the temporal array is copied back to the output array where Z-order is transformed to the default row-major order. The optimization essentially reduces the number of Z-order computation at the bitplane level to 1/K at the raster cell level. Our experiments have shown that this optimization is very effective in improving the overall system performance. While the decompression performance of our technique is inferior to zlib without the optimization using a single CPU-core, the performance order is reversed with the optimization. One disadvantage of the optimization is that, in a way similar to merge sort, a temporal array as large as the output array is required and copying the temporal array to the output array may require additional memory bandwidth. This might explains that, despite our decompression technique performs better than zlib using a single CPU core, it becomes inferior to zlib when all 16 cores are used, possibly due to memory bandwidth contention. However, we believe that further optimizations, for example, better reordering of accesses to raster cells and data pre-fetching, can potentially improve the decompression performance significantly for both serial and multi-core implementations of our BQ-Tree based decompression design. Similar improvements can also be applied to compression, although the compression performance of our technique is already significantly faster (4.1X) for serial implementation and considerably faster (36%) for 16-core parallel implementation using a chunk size of 1024*1024 than zlib, as reported in Section 4.2. In general, as our technique is likely to be memory bandwidth bound, more efficient data accesses can further significantly improve the performance for both compression and decompression.

3.5 Parallelization on Multi-Core CPUs

Although the parallel performance of our multi-core CPU implementation using a dual Intel Xeon 8-core E5-2650V2 machine with 16 CPU cores running at 2.6 GHZ has been mentioned earlier, we would like to present details of parallelization and provide some discussions in this section. For a large geospatial raster, a natural way to parallelize compression and decompression is to split it into multiple chunks and let each processing unit (i.e., a CPU core in this case) to process a chunk independently. Although we are aware of more sophisticated parallelization techniques might perform better with respect to load balancing (e.g., work stealing implemented in Intel Thread Building Blocks - TBB[11]), we have decided to use the simple "parallel for" parallelization directive (or pragma) for a simple implementation as we aim at a lightweight technique. Since most computing systems support OpenMP compliers and runtime systems, the extra overhead of parallelization is minimized. As discussed in the experiment section, we will also use OpenMP to parallelize zlib for compressing and decompressing large-scale rasters after they are chunked for fair comparisons.

During compression and decompression, each raster chunk will have its own input and output arrays as well as local variables. As such, the parallelization process is designed to be embarrassingly parallelizable. However, after all chunks are compressed, the output sizes need to be combined so that the compressed data can be written to the proper memory location and complete the final concatenation process. While the combination process can be easily implemented in parallel through a prefix-sum process (McCool et al 2012), since the number of chunks is typically limited and moving memory-resident data in large chunks is fairly fast on modern CPUs, we simply perform the final concatenation process sequentially with negligible overhead. We note that, it might be more beneficial to use a linked list to virtually concatenate these compressed chunks which can be effortlessly done on CPUs, instead of actually moving the chunks in memory. When the compressed chunks need to be written to disks, they can simply be output sequentially without concatenation as concurrent I/Os may perform worse in a personal computing environment. We note that the compressed chunks can naturally be used to support parallel I/Os in cluster computing environment by distributing the chunks into multiple disk strips (Lofstead et al 2011, Schendel et al 2012b, Lin 2013). This is left for our future work.

4 Experiments and Results

4.1 Data and Experiment Environment Setup

Although our technique can be applied to virtually any type of geospatial rasters, in this study, we will be focusing on NASA SRTM 30 meter resolution elevation data in the CONUS region. The SRTM data was obtained on a near-global scale in February 2000 from a space-borne radar system and has been widely used in many applications since then. The CONUS raster dataset has a raw volume of 38 GB and about 15GB when compressed in TIFF format in 6 raster data files. There are about 20 billion raster cells in these 6 files and we refer them collectively as the NASA SRTM raster hereafter. Like many geospatial rasters, cell values in this dataset are 16-bit short integers.

To accommodate computer systems with lower memory capacities, we have further split the 6 raster files into 36 tiles to ensure that each tile requires limited capacity (< 6GB). The original raster sizes and their partitions are listed in Table 1. Given that our experiment system has 16 CPU cores, such data partition also make it possible to roughly balance the workload at the raster chunk level using different chunk sizes. For example, there will be 42 chunks for raster file #1 using a chunk size of 4096*4096 and each core will process 2-3 chunks on average. Clearly, the smaller the chunk size and the larger

[11] https://www.threadingbuildingblocks.org/

number of chunks, the better chance for load balancing. However, this is at the cost of higher metadata and scheduling overheads.

The E5-2650V2 CPU running at 2.6 GHZ used in our experiments, which are released in the third quarter of 2013, are reasonably up-to-date. The memory bandwidth shared by the eight cores in a CPU is 59.7 GB/s. As we focus on compression and decompression performance (runtimes), we assume all data are memory resident before compression and decompression and we do not include disk I/O times. All programs are compiled with –O3 optimization level using gcc 4.9 and linked with zlib 1.2.3 when the "uncompress" or "compress" API in the library is used. We have also experimented two raster chunk sizes, i.e., 1024*1024 and 4096*4096, and we have obtained similar results. While we will report the overall results using the summations of the 36 raster tiles for both chunk sizes, due to space limit, we will only report the runtimes of individual tiles for the chunk size 1024*1024.

Table 1 List of SRTM Rasters, Split Schemas and # of Chunks

Raster file #	Dimension	Split Schema	#of Chunks in each raster tile	
			1024*1024	4096*4096
1	54000*43200	2*2	27*22	7*6
2	50400*43200	2*2	25*22	6*6
3	50400*43200	2*2	25*22	6*6
4	82800*36000	2*2	41*36	11*9
5	61200*46800	2*2	30*23	9*6
6	68400*111600	4*4	17*28	5*7
Total	20,165,760,000	36		

In addition to using zlib as a baseline for comparisons, we have also experimented a wavelet-based image compression package called Epsilon[12] whose data format is supported in the popular Geospatial Data Abstraction Library (GDAL[13]) through the Rasterlite[14] driver. Although Epsilon is designed to support 8-bit RGB images, we are able to make it work for 16-bit integer rasters by modifying the source code slightly. Unfortunately, it seems that the version of Epsilon we use supports lossy compression only. Although our experiments have shown that the average error of most chunks are in the order of 1-2 in most raster chunks, which is only a small fraction of the average/maximum of the elevation values, the maximum error can be very large in some chunks, likely due to the no-data values in these chunks (as discussed in Section 1). Although we do not believe such lossy compression is acceptable in practical applications, we include its compression and decompression runtimes for comparisons to provide an idea of the practical performance of wavelet based techniques for the NASA SRTM raster dataset.

[12] http://sourceforge.net/projects/epsilon-project/
[13] http://www.gdal.org/
[14] http://www.gdal.org/frmt_rasterlite.html

4.2 Overall results

The experiment results for our BQ-Tree (bqtree), Zlib/LZ77 (zlib) and Epsilon (epsilon) based techniques are listed in Table 2, where RT stands for runtime (in milliseconds). From Table 2 we can see that, our BQ-Tree technique is significantly faster than zlib for compression. The speedup is about 4.1x for the serial implementation (1264697/311746) and 6.5x for 16-core implementation (157341/24120), respectively, for chunk size 1024*1024. The speedups change moderately for chunk size 4096*4096, which are 3.5x and 4.5x for the serial implementation and the 16-core implementation, respectively. On the other hand, for decompression, while our BQ-Tree technique is considerably faster (124442/91794 - 1=36%) for the serial implementation for chunk size 1024*1024 and slightly faster (124445/117289-1=6%) for chunk size 4096*4096, the 16-core implementation is 73% slower (22318/12859 - 1) and 124% slower (28857/12882 - 1) for the chunk size 1024*1024 and 4096*4096, respectively. Clearly the asymmetric design of the underlying LZ77 algorithm makes zlib 10X+ faster for decompression than for compression. Although our BQ-Tree technique is also 3X faster for decompression than for compression in the serial implementation for both chunk sizes, the asymmetry is mostly due to implementation and hardware characteristics, not algorithmic design.

Table 2 Compression/Decompression Runtimes (in milliseconds) of Three Techniques Using Two Chunk Sizes

Chunk size	1024*2024		4096*4096	
serial/parallel grouping	RT-serial	RT-16-core	RT-serial	RT-16-core
bqtree-compression	311746	24120	365220	35046
bqtree-decompression	91794	22318	117289	28857
zlib-compression	1264697	157341	1264959	156495
zlib-decompression	124442	12859	124445	12882
epsilon-compression	1528558	204905	1788340	300959
epsilon-decompression	1204015	172251	1291030	242197

Comparing the runtimes of the serial implementations and the 16-core implementations, while the speedup for bqtree-compression, zlib-compression and zlib-decompression are 13.0X, 8.0X and 9.7X, respectively, the speedup for bqtree-decompression is only 4.1X, using chunk size 1024*1024. Similar results can be obtained for chunk size 4096*4096. It is also interesting to see that the speedups of the 16-core implementation for both compression and decompression of our BQ-Tree technique drops as chunk size increases. In particular, the speedup of compression is 13X for chunk size 1024*1024 while it drops to 10.4X for chunk size 4096*4096. The speedup also drops slightly for decompression (from 4.11X to 4.06X), possibly because the speedup is already low. On the other hand, the speedups remain very stable for zlib, which is about 8.0X for compression and 9.7X for decompression, for both chunk size of 1024*1024 and 4096*4096.

The relatively lower performance and lower speedup of the 16-core implementation of our BQ-Tree technique motivate us to investigate the interaction between our algorithms and the multi-core CPU architecture of the experiment machine. We attribute the lower performance to memory contentions when all 16 cores are fully utilized for parallel decompression. Despite the memory bandwidth on the Intel Xeon E5-

2650V2 CPU (59.7 GB/s) in our experiment system is significantly higher than previous generation CPUs, it is still much lower than GPUs (up to ~300 GB/s). Since the 8 cores in a CPU share the memory bandwidth, the share of each core is very limited. We expect that high CPU memory bandwidths are likely to improve the performance of our BQ-Tree technique for decompression, when memory bandwidth is not a limiting factor. On the other hand, when comparing the BQ-Tree based compression and decompression, the compression has much better scalability (10-13X using 16 cores) than decompression (~4X). While we are still in the process of fully understanding the realized implementation asymmetry (although our design is conceptually symmetric from an algorithmic perspective), we suspect that the fact is due to a non-optimized memory access pattern in the decompression implementation. While a raster cell is only accessed once from memory and its value can be kept in a register during compression, the raster cell may need to be accessed K times during decompression to combine bits at the K bitplanes. Since it is non-trivial to synchronize decompressing across K bitplanes and keep the resulting raster cell in a register, we leave such potential optimization for future work.

We next turn to analyzing the overall performance by including compression and decompression and their use frequencies into consideration. First of all, given the measured runtimes of compression and decompression for a particular technique (BQ-Tree or zlib) using a specific implementation (serial or 16-core) and a chunk size (1024*1024 or 4096*4096), assuming the compression and decompression are equally frequent, the total runtime can be easily computed and compared. Despite that our BQ-Tree is slower than zlib for the 16-core implementation, when the frequencies of compressions and decompressions are the same, overall, the BQ-Tree technique is 3.66X and 3.44X better than zlib, for the serial and 16-core implementations using a chunk size of 1024*1024, respectively. The speedups are 2.88X and 2.65X using a raster size of 4096*4096. Second, when comparing the overall performance of zlib and BQ-Tree, assuming that the frequency of compression is p and the frequency of decompression is (1-p) and the compression and decompression times for zlib and BQ-Tree are t11, t12, t21 and t22, for a specific implementation using a specific chunk size, in order for zlib to perform better than BQ-Tree, we need $t11*p+t12*(1-p)<t21*p+t22*(1-p)$. This would require $p<(t12-t22)/(t12-t11+t21-t22)$. Since the performance orders for compression and decompression are different in the 16-core implementation, by plugging the measured t11, t12, t21 and t22 runtimes of the implementation, it is easy to compute that p needs to be less than 6.6% using 1024 chunk size in order for zlib to perform better than BQ-Tree. The requirement is quite strong from a practical perspective, i.e., decompression frequency needs to be at least $(1-p)/p=14.1$ time higher than compression. Similarly, zlib performs better only if $p<11.6\%$ in the 16-core implementation using a chunk size of 4096*4096 which requires decompression to be 7.6 time higher than compression. Note that since BQ-Tree performs better than zlib for both compression and decompression in the serial implementations, BQ-Tree always has a better overall performance than zlib regardless of p. This can be verified by computing p values from t11, t12, t21 and t22 for the serial implementations using both chunk sizes. The computed p values are negative which are impossible.

A shown in the last two rows of Table 2, when comparing with the runtimes of the Epsilon wavelet based compression and decompression with these of the BQ-Tree technique, we can see that the Epsilon implementation is about 8X slower than our BQ-Tree technique for the 16-core implementation using chunk size 1024*1024 and 4096*4096, respectively. For the serial implementation for compression, our BQ-Tree is about 5X faster for both chunk sizes. The speedups of BQ-Tree over Epsilon in the serial implementation for decompression are 13X and 11X for chunk size 1024*1024 and 4096*4096 respectively. While both BQ-Tree and Epsilon are designed to be symmetric, the results clearly demonstrate that BQ-Tree has a much lower computation complexity than Epsilon. On the other hand, although we are targeting at lossless compression in this study, the compressed sizes of Epsilon are about 2.2 GB, which is considerably lower than both BQ-Tree and zlib. The high compression ratios might be suitable for certain applications that can tolerate low average errors and rare but high absolute errors. However, this is out of the scope of this study.

4.3 Results of individual raster tiles

The results reported in Section 4.2 are based on the summations of all the 36 raster tiles from the 6 raster files as listed in Table 1. To further understand the distributions of compressed sizes of our BQ-Tree technique, the three components of the resulting compressed bytes are plotted in Fig. 5. The metadata sizes are proportional to chunk sizes as we need to record lengths of the resulting quadtree arrays and LLQS arrays so that processing units can access the respective chunks in parallel during decompression. From Fig. 5, it is can be seen that the metadata sizes are negligible and the LLQS sizes dominate. The results may suggest that further compressing LLQS arrays can be beneficial. Fig. 6 compares the compressed sizes of our BQ-Tree technique and the zlib. It can be seen that the sizes are comparable with BQ-Tree achieving slightly better compression ratios than zlib in some cases and zlib performs better in some other cases. The results explain why the two techniques (BQ-Tree and zlib) have roughly the same overall compression ratio for chunk size 1024*1024.

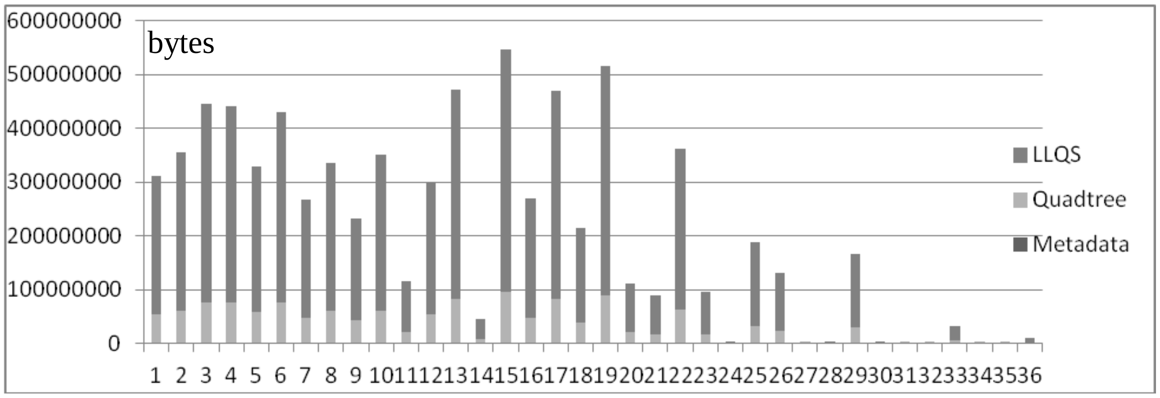

Fig. 5 Compressed Sizes for the 36 Raster Tiles of the BQ-Tree Approach

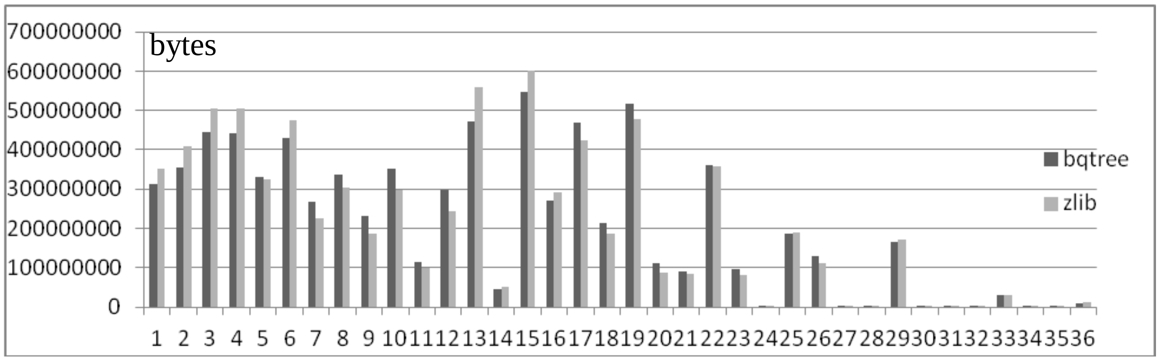

Fig. 6 Comparisons of compressed sizes of 36 raster tiles using BQ-Tree and zlib

The runtimes of the serial and the 16-core implementations of both compressions and decompressions of BQ-Tree technique are plotted in Fig. 7 through Fig. 10. It can be seen that our technique compresses much better for both the serial and the 16-core implementations. While zlib vary significantly and performs much faster for raster tiles that have large portions of no-data values (especially for those that are in the border of the CONUS region where ocean raster cells are marked as no-data). In contrast, our technique is largely data-independent, which can be both advantageous and disadvantageous. The data-independent feature of the BQ-Tree technique can be advantageous for datasets whose raster cells vary significantly, although it may need to do more work than zlib for datasets whose cells are uniform (e.g, large portions of raster cells have no-data values, as in several among the 36 raster tiles). While it is straightforward to construct some simple data structures with near linear workload (e.g., min-max quadtree discussed in Zhang et al. 2010, Zhang et al. 2013) and use them to guide choosing compression algorithms, we leave a more comprehensive study for future work. We note that, despite the performance of our BQ-Tree technique and zlib are very close for raster tiles with large portions of no-data values in both the serial implementation and the 16-core implementation, our technique is much faster for other tiles as shown in Fig. 7 and Fig. 8. This may explain the overall performance for compression presented in Section 4.2.

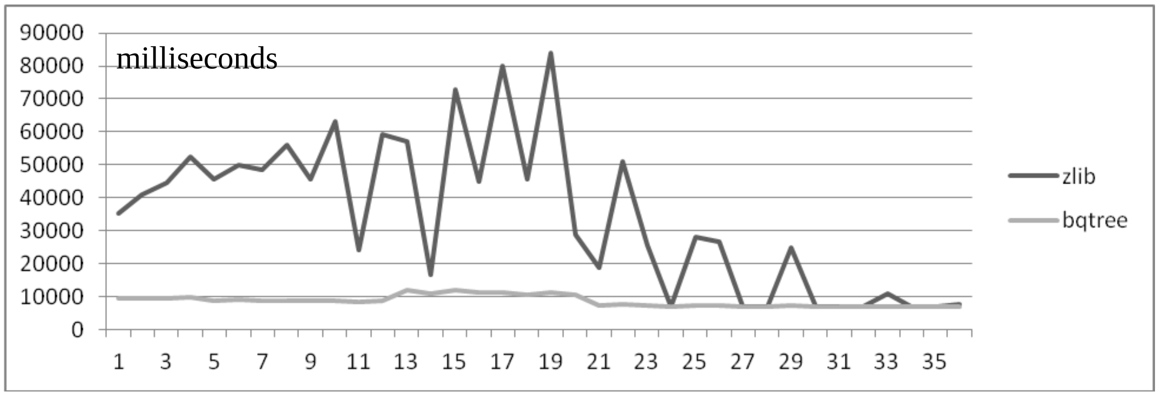

Fig. 7 Compression time plots for serial zlib and BQ-Tree implementations

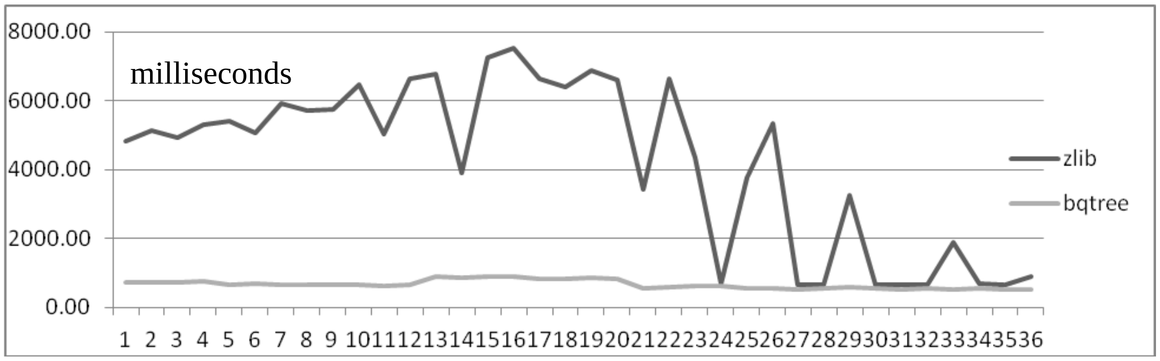

Fig. 8 Compression time plots for the 16-core zlib and BQ-Tree implementations

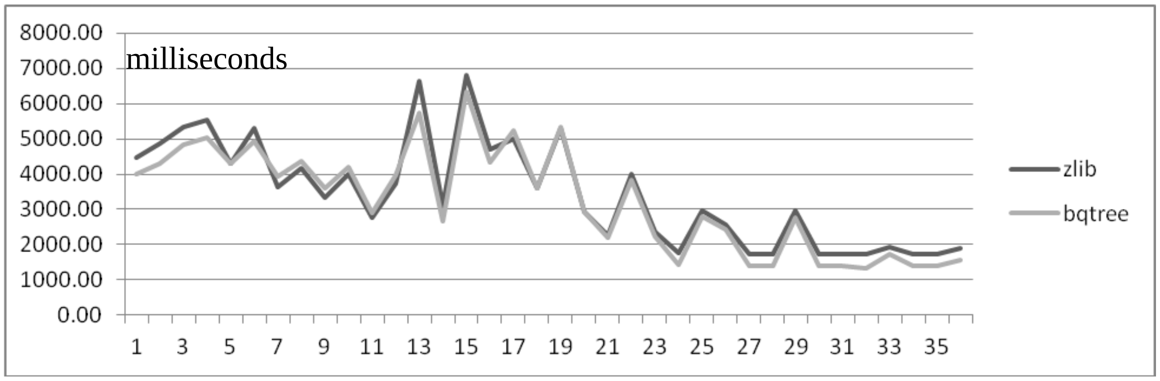

Fig. 9 Decompression time plots for the serial implementation

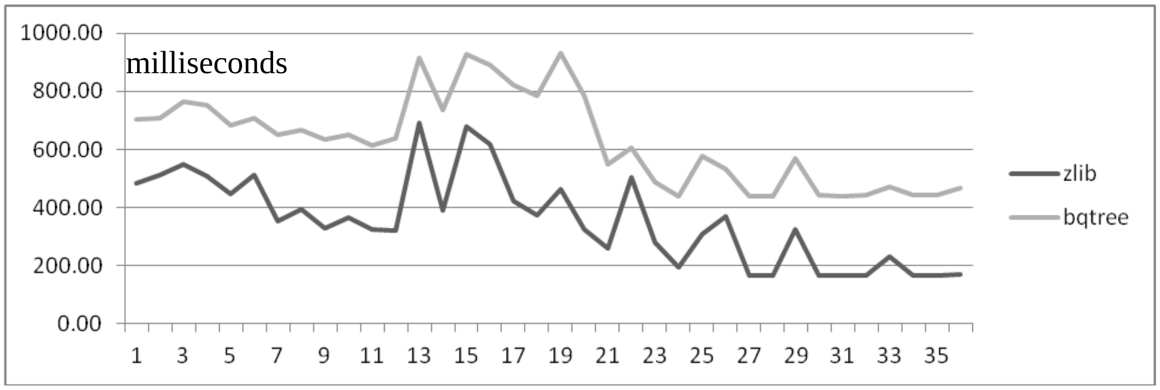

Fig. 10 Decompression time plots for the 16-core implementation

The plots in Fig. 9 and Fig. 10 tell a somewhat different story in the multi-core CPU implementations. Although BQ-Tree is overall 36% faster than zlib in the serial implementation for decompression, as shown in Fig. 9, there are many cases that zlib is actually faster. Furthermore, the BQ-Tree technique performs worse in all cases for the 16-core implementation which explains the overall 1.7X slow down. As discussed in Section 4.2, memory bandwidth contention in the 16-core implementation might be an important reason. The different comparison results between the serial implementations and the 16-core implementations may suggest the importance of the interaction between algorithmic design and hardware architecture, rather than algorithm or CPU alone, in determining realizable performance. While we leave further optimizations of our decompression algorithms for future work as discussed previously, we would like to stress that, similar to analyzing the aggregated overall performance, our BQ-Tree technique performs several times better than zlib for most of the 36 raster tiles, unless the decompression is much more frequently applied than compression. Since we target at the applications that have balanced compression and decompression, our BQ-Tree technique performs better than zlib when overall performance is measured, in addition to being designed to be simple and portable.

4.4 Comparisons with GPU-based Decompression

We also would like to compare the 16-core CPU implementation for decompression with the GPU-based decompression technique reported in (Zhang et al 2011). By running the GPU-based implementation using the same dataset on Nvidia Quadro 6000 (based on Fermi architecture) and Nvidia GTX Titan (based on Kepler architecture) devices using the 4096*4096 chunk size, we measure the runtimes for decompression which are 16.2s and 8.3s on the two devices, respectively. Note that the GPU-based compression is currently unavailable which prevents us from performing more comprehensive comparisons. However, the results do show that, while our 16-core CPU implementation is still slower (28.9s), the multi-core CPU and the GPU implementations are comparable in decompression performance. The higher performance on GPUs can be attributed to much larger number of processing units (448 cores for the Quadro 6000 device and 2688 core for the GTX Titan device) and higher memory bandwidth (144 GB/s for Quadro 6000 and 288 GB/s for GTX Titan), in addition to some implementation specific differences.

While the result may lead us to conclude that GPU-based decompressions are better, we argue that, transferring compressed data to GPUs and transferring uncompressed data from GPUs may cost roughly about the same time as decompression itself, given that effective PCI-E bandwidth is generally limited to 4-8 GB/s. Transferring the 7.2 GB compressed data to GPUs and the 38 GB decompressed data back to CPUs may well take 5-10 seconds. As a result, our pure multi-core CPU-based implementation is competitive when the decompressed data needs to be subsequently processed in CPUs, in addition to be much simpler in implementation. We believe it is more appropriate to determine whether to use multi-core CPUs or GPUs for decompression based on the destinations of uncompressed data when data transferring between CPUs and GPUs plays an important role in overall performance.

Meanwhile, we would also like to bring to the attention that, while GPUs are known to efficiently exploit Single Instruction Multiple Data (SIMD) computing power, modern CPUs are also equipped with Vector Processing Units (VPUs) (Hennessy and Patterson, 2012) . The SIMD widths have steadily increased from 2-wide (MMX) to 4-wide (SSE) and 8-wide (AVX). This makes it attractive to utilize SIMD computing power on CPUs. Although neither the zlib nor our BQ-Tree technique uses SIMD computing in this study, we expect that the CPU+VPU implementations can potentially be competitive or are able to achieve even better performance than the current generation GPUs. Although traditionally it is difficult and error-prone and thus unproductive to exploit VPUs by using SIMD intrinsic functions, the newly available ISPC[15] compiler allows developer to write scalar code in a way similar to CUDA programming and make use of VPUs easier and more productive (Pharr and Mark 2012). We leave the interesting research topic to our future work.

5 Conclusions and Future Work

In this study, we aim at developing a lightweight lossless data compression technique for large-scale geospatial rasters to support efficiently streaming data from disks to CPU memories as well as among distributed computing nodes and file systems. Despite that the idea on using quadtrees to encode images and rasters is quite simple and has been exploited before, to the best of our knowledge, we are the first to demonstrate that the quadtree based technique can perform several times better than the popular generic zlib technique not only for the serial implementation but also the overall performance for the multi-core implementation. Given that our implementations are still under intensive optimizations, we are optimistic in achieving comparable or even better performance than zlib for decompression implementation after reducing memory bandwidth contentions in the multi-core CPU implementation. The simple design and implementations of our technique make it easier to port the technique to new hardware, including VPUs on CPUs, GPUs and Intel Xeon Phi devices when compared with zlib.

For future work, in addition to further optimizations for multi-core CPUs (e.g., data layouts and access patterns for cache efficiencies) and porting the designs and implementations to new parallel platforms as discussed inline, we would like to perform more experiments on more large-scale rasters and compare our technique with additional data compression techniques to understand its strengths and weaknesses. Another research direction is to explore different ways in assembling and disassembling rasters from and to bitmaps for BQ-Tree coding, in addition to the natural bitplane bitmaps that are used in this study. Finally, while we have opted for simplicity in this study, it is interesting to explore the tradeoffs among computation workloads, compressed sizes and memory bandwidth utilizations.

ACKNOWLEDGEMENT

This work is supported in part through NSF Grants IIS-1302423.

References

1. Chou, J., Wu, K., Prabhat (2011). FastQuery: A Parallel Indexing System for Scientific Data. Proceedings of IEEE International Conference on Cluster Computing (CLUSTER'11), pp. 455-464.
2. Durdevic, D.M. and Tartalja, I. I. (2013). HFPaC: GPU friendly height field parallel compression. GeoInformatica, 17(1), pp. 207-233.
3. Gong,Z., Boyuka, D. A. et. al (2013). PARLO: PArallel Run-Time Layout Optimization for Scientific Data Explorations with Heterogeneous Access Patterns.

[15] http://ispc.github.io/

Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID'13), pp.343-351.

4. Gong, Z., Lakshminarasimhan, S., et al. (2012). Multi-level Layout Optimization for Efficient Spatio-temporal Queries on ISABELA-compressed Data. Proceedings of IEEE Conference on Parallel & Distributed Processing (IPDPS'12), pp. 873-884.
5. Gonzalez-Conejero, J., Bartrina-Rapesta, J.; Serra-Sagrista, J. (2010), JPEG2000 Encoding of Remote Sensing Multispectral Images With No-Data Regions, IEEE Geoscience and Remote Sensing Letters, , 7(2), pp 251-255.
6. Gosink, L. J., Shalf, J., et al. (2006). HDF5-FastQuery: Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices. Proceedings of Scientific and Statistic Database Management (SSDBM'06), pp.149-158.
7. J. Hennessy, D. A. Patterson (2011). Computer Architecture: A Quantitative Approach (5th ed.), Morgan Kaufmann, Waltham, MA, USA.
8. Jenkins, J., Arkatkar, I., et al. (2013) ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying. Transaction on Large Scale Data and Knowledge Centered Systems, vol. 10, pp. 95-114.
9. Kanov, K., Burns, R.C., et al. (2012) Data-intensive spatial filtering in large numerical simulation datasets. Proceedings of ACM Supercomputing Conference (SC'12), #60.
10. Lakshminarasimhan, S., Boyuka D. A., et al. (2013a). Scalable in situ scientific data encoding for analytical query processing. . Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'13), pp. 1-12.
11. Lakshminarasimhan, S., Shah, N., et al. (2013b). ISABELA for effective in situ compression of scientific data. Concurrency and Computation: Practice and Experience, 25(4), pp. 524-540.
12. Lemire, D. and Boytsov, L. (in press). Decoding billions of integers per second through vectorization. Software: Practice and Experience.
13. Lin, K.-W. , Byna, S., et al (2013). Optimizing FastQuery performance on Lustre file system. Proceedings of Scientific and Statistic Database Management (SSDBM '13), #29 .
14. Liu, J., Crysler, B., et al. (2013a). Locality-driven high-level I/O aggregation for processing scientific datasets. Proceedings of IEEE BigData Conference (BigData'13), pp. 103-111.
15. Liu, Z., Wang, B., et al. (2013b). Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application. Proceedings of international Conference on Computer Communications and Networks (ICCCN'13), pp. 1-7.
16. Lindstrom, P. and Cohen, J.D. (2010). On-the-fly decompression and rendering of multiresolution terrain. Proceedings of the ACM symposium on Interactive 3D Graphics and Games (SI3D'10), pp. 65-73.
17. Lofstead, J.F., Polte, M., et al. (2011). Six degrees of scientific data: reading patterns for extreme scale science IO. Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'11), pp.49-60.
18. McCool, M., Robison, A.. and Reinders, J. (2012). Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann.
19. Morton, G.M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. IBM Technical report.

20. Ozsoy, A., Swany, M., Chauhan, A. (2014). Optimizing LZSS compression on GPGPUs. Future Generation Computer Systems, vol. 30, pp. 170-178.
21. Peter, J., Straka, M., et al. (2013).Petascale WRF Simulation of Hurricane Sandy Deployment of NCSA's Cray XE6 Blue Waters. Proceedings of ACM Supercomputing Conference (SC'13), #63.
22. Pharr, M. and Mark, W. (2012). ISPC: A SPMD compiler for high-performance CPU programming. Proceedings of Innovative Parallel Computing (InPar).
23. Raman, R. and Wise, D.S., 2008. Converting to and from Dilated Integers. IEEE Transactions on Computers, 57(4), 567-573.
24. Salomon, D. (2006). Data Compression: The Complete Reference (4th edition). Springer.
25. Schendel, E. R., Jin, Y., et al. (2012a): ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression. Proceedings of IEEE International Conference on Data Engineering (ICDE'12), pp.138-149.
26. Schendel, E. R., Pendse, S. V., et al. (2012b). ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'12), pp. 61-72.
27. Schmit, T.J., Li, J., et al. (2009). High-spectral- and high-temporal resolution infrared measurements from geostationary orbit. Journal of Atmospheric and Oceanic Technology, 26(11), pp. 2273-2292.
28. Sinha, R. R., Winslett, M., Wu, K. (2009) Finding Regions of Interest in Large Scientific Datasets. Proceedings of Scientific and Statistic Database Management (SSDBM'09), pp.130-147.
29. Soroush,E., Balazinska, M., et al. (2011). ArrayStore: a storage manager for complex parallel array processing. Proceedings of ACM Conference on Management of Data (SIGMOD'11), pp. 253-264.
30. Tian, Y., Klasky, S., et al. (2013): DynaM: Dynamic Multiresolution Data Representation for Large-Scale Scientific Analysis. Proceedings of IEEE Conference on Networking, Architecture and Storage (NAS'13), pp. 115-124.
31. Wu, K., Shoshani, A. and Stockinger, K. (2010). Analyses of multi-level and multi-component compressed bitmap indexes. ACM Transaction on Database Systems (TODS), 35(1), #2.
32. Wu, K., Otoo, E.J. and Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. ACM Transaction on Database Systems (TODS), 31(1), pp.1-38.
33. Wu, K., Otoo, E., Shoshani, A. (2004). On the performance of bitmap indices for high cardinality attributes. Proceedings of Conference on Very Large Data Bases (VLDB'04), pp. 24-35.
34. Zhang, J., You, S. and Gruenwald, L. (2011). Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs. Proceedings of the 19th ACM conference on Advances in Geographic Information Systems (GIS'11), pp. 457-460.
35. Zhang,J. and You, S. (2013) High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. International Journal of Geographical Information Science, 27(11), pp. 2207-2226.

36. Zhang,J. and You, S. (2010) Supporting Web-Based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree. Proceedings of Scientific and Statistic Database Management (SSDBM'10), pp.379-396.
37. Zheng, F., Zou, H., et al. (2013). FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. Proceedings of IEEE Conference on Parallel & Distributed Processing (IPDPS'13), pp. 320-331.