# Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on Multicore CPUs and 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. In addition to utilizing the chunk-level coarse grained parallelism on both multicore CPUs and GPGPUs, we have also developed two fine-grained parallelization schemes and their four implementations by using different system and application level optimization strategies. Experiments show that the best 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 6X speedup when compared with the best dual quadcore CPU implementation and 239-290X speedups when compared with a baseline single thread CPU implementation.

## 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[1][2], quadtree probably is the most popular family due to its effectiveness and simplicity in indexing, compressing and querying both vector and raster geospatial data. For example, quadtree indexing of polygons has been implemented in Oracle Spatial [3] and a variant of quadtree representation has been implemented in Microsoft SQL Server Spatial [4]. Quadtrees have also been used to encode images [5][6]. Bitplane level quadtree coding/indexing is also strongly related to bitmap indexing [7] when applied to raster geospatial data. Many bitmap based efficient query processing techniques can readily be applied to raster geospatial data to improve query responses and reduce I/O accesses. Reducing disk and memory I/O overheads is especially beneficial to modern parallel processors given that I/O, rather than computation, is increasingly becoming the primary bottleneck of overall system performance [8][9]. In addition, 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. Unfortunately, research on cache-conscious spatial data structures and algorithms has received very little attention. The performance of classic spatial data structures (such as quadtrees) on modern commodity parallel processors with different configurations of memory hierarchies is largely unknown.

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 parallel computing resources available in commodity personal computers in decoding bitplane quadtrees, including both multicore CPUs and GPGPUs. 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 [10]) of high-resolution, time-evolving and multi-variant raster geospatial data to effectively support global environmental studies. More specifically, our technical contributions in this study are outlined as follows:

(1) We have designed an efficient and cache-friendly quadtree data structure, termed as Bitplane Quadtree (BQ-Tree), for coding bitplane bitmaps of raster geospatial data. The BQ-Tree orders tree nodes in a breadth-first traversal manner which does not need pointers to chain parent-child node pairs. The data structure can be naturally serialized and works with both CPUs and GPGPUs efficiently.

(2) We have proposed two different parallelization schemes to decode BQ-Trees on GPGPUs. We have also developed optimization strategies for each of the two approaches with demonstrated efficiencies.

(3) We have conducted comprehensive performance comparisons on both multicore CPUs and GPGPUs using a real NASA satellite remote sensing dataset. Experiments show that the best GPGPU implementation can achieve nearly 6X speedup running on an Nvidia C2050 card when compared with the best multicore CPU implementation running on an Intel dual quadcore CPU. The best GPGPU implementation also achieves 239X-290X speedups over a baseline single-threaded CPU implementation.

The rest of the paper is organized as follows. Section 2 introduces background and research motivations. Section 3 presents the BQ-Tree data structure. Section 4 presents the encoding and decoding algorithms for BQ-Trees on CPUs and discusses parallelization of decoding on both CPUs and GPGPUs. Section 5 provides two fine-grained parallelization schemes on

GPGPUs and its four implementations. Section 6 is the performance comparisons. Section 7 briefly introduces related works. Finally, Section 8 is the conclusion and future works.

## 2. BACKGROUND AND MOTIVATIONS

Raster data representation is a major data model for geospatial data [11]. Surprisingly, compared to vector geospatial data that hundreds of indexing techniques have been developed [1][2], 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 [12] provides a comprehensive framework to manage multidimensional arrays, including raster geospatial data. While the current implementation (Version 0.75) does support generic compression methods (by using Zlib and BZlib), 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 (4/8 bytes) to data (1 bit) ratio when applied to bitplane bitmaps of rasters.

Bitmap indexing has been extensively investigated in relational databases [7][13][14][15][16][17][18]. While virtually all bitmap indexing techniques can be applied to raster geospatial data by ordering raster cells into a one dimensional sequence based on a spatial order, e.g., row-major, column-major, Z-order and Hilbert Space Filling Curve (SFC) [19], they are not designed for raster geospatial data. Unlike quadtree base query processing that naturally returns spatial hierarchy of resulting raster cells, queries based on classic bitmap indexing can only return individual tuple (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. Given that bitmap indexing has been widely used in commercial relational database systems and open source implementations (e.g., FastBit [20]) are available, we next discuss how 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.

According to [7], bitmap indexing technologies can be divided into three categories, namely binning, encoding and compression. Binning is to produce a set of identifiers (e.g., bin numbers) from a set of arbitrary values to be used in the encoding step. This step is optional in indexing raw remotely sensed data as the raster cell values are usually already binned when optical or electronic signal strengths are converted to digital numbers. The encoding step takes the bin identifiers and translates them into a set of bitmaps. As detailed in [7], there are three major types of encoding schemes, namely equality encoding, range encoding and interval encoding. They are suitable for different types of queries. More complex encoding schemes can be derived by combining the three encodings within a multi-component and multi-level framework. The approach that we use for encoding raster cells in this study can be considered as a multi-component encoding that uses N components, where N is the number of bits for the raster to be encoded. Each component represents a binary raster and the $i^{th}$ binary raster consists of all the $i^{th}$ bits of the raster cells, i.e., bitplane bitmap. All components are encoded the same way using the basic exact encoding scheme as each component value is either 0 or 1. The encoding scheme is called binary encoding in bitmap indexing of relational data and we term it as bitplane bitmap encoding when it is applied to geospatial rasters. With the mapping between bitplane bitmap coding of raster geospatial data and the generic bitmap coding of relational data, all the query techniques that utilize bitmap indexing can now be applied to speed up queries on individual raster cells.

Our plan is to replace existing bitmap compression techniques (the third step of bitmap indexing), such as run-length and Word-Aligned Hybrid (WAH) [16] that are spatial agnostic and utilize flat data structures, with the BQ-Tree encoding 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. Utilizing parallel hardware, including multicore CPUs and GPGPUs, to speed up the reconstructions is a promising solution and is the focus of the paper.

Compared to CPU computing, GPGPU computing is relatively young. We next briefly introduce the basics of GPGPU computing to help understand the GPGUP decoding algorithms and implementations to be presented in Section 5. A Graphics Processing Unit (GPU) is a hardware device that is originally designed to work with CPU to accelerate rendering of 3D or 2D graphics. The highly parallel structures of modern GPU devices make them more effective than general-purpose CPUs for a range of complex graphics-related algorithms. The concept of General Purpose GPU computing turns the massive floating-point computational power of a modern graphics accelerator's graphics-specific pipeline into general-purpose computing power. GPGPU computing technologies have gained considerable interests in many scientific research areas in the past few years [21][22]. Currently, Nvidia Compute Unified Device Architecture (CUDA) [23] might be the most popular parallel development framework on GPGPUs [24][25].

While different models of GPU devices have different configurations and parallel processing capabilities, CUDA-enabled GPU devices are organized into a set of Stream Multiprocessors (SMs). Each SM has a certain number (e.g., 16 or 32) of computing cores. All the cores in a SM share a limited amount of fast memory called shared memory (with a few cycles delays) and all the SMs have access to a large but slow pool of global memory on the device (with a few hundreds of cycles delays). According to CUDA, developers write special C-like code segments called kernels. The kernels are invoked by the associated CPU code to run on GPU devices. CUDA based GPGPU programming makes it easier for task and data

decomposition and subsequent parallel computing. Basically a developer specifies the size of the layout of the data to be processed in the units of data blocks and the number of threads to be launched inside a data block. The GPU device is responsible for mapping the data blocks to the computing blocks within the SMs through hardware-based scheduling which is transparent to developers/users. Since each SM has limited hardware resources, such as the number of registers, shared memory and thread scheduling slots, a SM can accommodate only a certain number of blocks subjected to the combination of the constraints. Carefully selecting block sizes allows a SM to accommodate more blocks simultaneously and, subsequently, improves parallel throughputs.

## 3. THE BQ-TREE DATA STRUCTURE

Given a bitplane bitmap of a raster R of size N*N (assuming $N=2^n$), as illustrated in Fig.1, the bitmap 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 [26][27]. To overcome these problems, we have designed a spatial data structure called BQ-Tree to efficiently represent bitmaps of bitplanes of a geospatial raster.

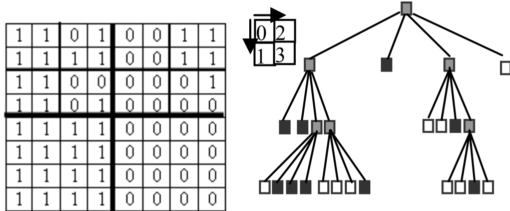


Fig. 1 Quadtree Representation of a bitplane bitmap

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 [19]. 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. While the details on the sequenced tree node array and the last level quadrant signature array (as well as the correspondence between the leaf nodes and the last level quadrant signatures) will be detailed next, we would like to mention that sequencing quadtree nodes and quadrant signatures as one-dimensional arrays is not only cache friendly but also makes it more interoperable between CPUs and GPUs that currently have distinct memory spaces in the respective devices.

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.

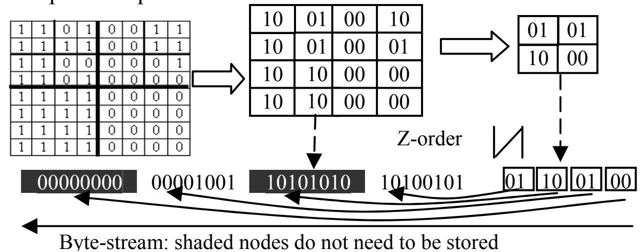


Fig. 2 Streaming BQ-Tree Nodes

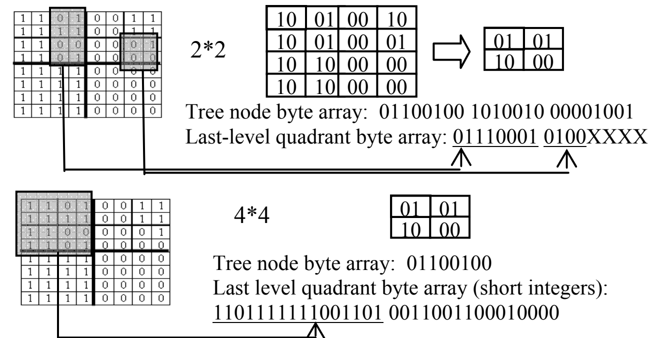


Fig. 3 Generating LLQS Array Using Different Quadrant Sizes

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. When k=1, half of the memory for storing BQ-Tree leaf nodes can be saved, as only 4 bits, instead of 8 bits, are needed for a 2*2 quadrant. Similar to compacting BQ-Tree nodes, LLQSs can also be compacted when they are all 0s (with "00" signatures in BQ-Tree leaf nodes) or when they are all 1s (with "10" signatures in BQ-Tree leaf nodes).

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. 3 shows the processes of compacting LLQSs using both 2*2 and 4*4 quadrant sizes. The bitmap shown in the left part of Fig. 3 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. 3) 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. 3) 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). Note that the bit order of last level quadrants (in both 2*2 and 4*4 cases) follows row-major order instead of Z-order to minimize Z-order calculation. Our experiments have shown that Z-order calculation can be expensive when not optimized. 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. CODING RASTER GEOSPATIAL DATA ON CPUs

Before presenting the GPGPU algorithms to decode large-scale raster geospatial data in Section 5, we next provide the baseline encoding and decoding algorithms on CPUs. The CPU encoding implementation provides encoded data for experimentation on both CPU and GPGPU based decoding algorithms. The single-thread CPU decoding implementation is used to validate the GPGPU decoding results, in addition to form the baseline for comparisons.

## 4.1 Encoding

The inputs for the encoding 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 B, 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 $B^* (1+4+16+\ldots+2^{m-q-1}*2^{m-q-1}) = B*(4^{m-q}-1)/3$ and the size of LA should be $B*2^{m-q}*2^{m-q}$. The encoding algorithm is given in Fig. 4 which is straightforward to follow by using the examples 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 either a byte or short integer stream by keeping only 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, depending on the last level quadrant sizes (Step 4). We note that it is possible to use Hamming distance [28] to define the "uniformity" of the last level quadrants by comparing their signatures with sequences of all 0s and all 1s. The approximation can reduce memory footprint of the LLQS array and may be desirable in many cases.

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 number of the quadrant
1.2 For each of the B 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 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 B 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
3.1 Skip all bytes whose values are 0x00 or 0xaa
Step 4 For all the elements in LA
4.1 If the last level quadrant size is 2*2 (q=1): skip all half-bytes with value of 0x0 or 0xF and combine two consecutive half bytes into one byte
4.2 If last-level quadrant size is 4*4 (q=2): skip all short integer values (2 bytes) of 0x0000 or 0xFFFF.

Fig. 4 BQ-Tree CPU Encoding Algorithm

## 4.2 Decoding

The decoding process is the reverse of the encoding process. A detailed procedure similar to Fig. 4 can be easily constructed and is skipped due to space limit. Instead, we next present an overview of the steps of the decoding algorithm that serves as the common base for both CPU and GPGPU implementations. 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 (depending on the quadrant signatures in the leaf nodes), or with the values in the LLQS array. 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|=(1<<i).

## 4.3 Discussions on Parallelization

While a formal proof is omitted due to space limit, we would like to note that both the encoding and decoding algorithms are data independent which means that the space and time complexities of the algorithms do not depend on data distributions. The relevant data are processed bitplane by bitplane and tree level by tree level, all on regular data structures (matrices and arrays). While the algorithms may not be the most work efficient ones when compared to quadtrees that allow depth-first traversals where certain quadrants can be skipped in encoding and decoding, the streamline processing feature of the proposed algorithms makes them cache-friendly and can potentially lead to more efficient implementations on modern hardware architectures that depend on deeper memory hierarchies and are sensitive to caching.

More importantly, the algorithms lend themselves to both coarse-grained and fine-grained parallel implementations. Given that current generations of commodity desktop computers or computing nodes of cluster computers typically have 4-12 CPU cores, it is natural to assign a chunk (e.g., 1024*1024) of a raster or a bitplane of a raster to a CPU core for coarse grained parallelization. We note that both the number of chunks c (assuming $c=2^p$ where p>=4) and the number of bitplanes (typically 8/16/32) are multiplications of the numbers of CPU cores in a commodity computer (typically 2/4/8). This can be implemented using quite a few parallel computing frameworks on CPUs, such as Pthreads, OpenMP, MPI for multicores, Phoenix (a MapReduce variation for multicore CPUs) [29], Intel Thread Building Blocks (TBB) and Microsoft Parallel Pattern Library (PPL). In this study, we have chosen to use OpenMP as the directive-based programming framework is easy to use and well supported by major compliers on both Linux and Windows platforms.

As the current commodity GPGPUs can launch much larger numbers of threads (hundreds or more) where groups of threads are executed in a SIMD (Single Instruction Multiple Data) manner, it is also important to explore fine-grained parallelization to coordinate GPGPU threads when processing bitplane bitmaps. While we have implemented both encoding and decoding algorithms on CPUs using chunk-level coarse-grained data parallelism, since our focus in this study is decoding (for the reasons discussed previously), we have only implemented the decoding algorithms on GPGPUs while leaving implementations of the encoding algorithms on GPGPUs for future work. Since encoding and decoding rasters based on the BQ-Trees are symmetrical, the GPGPU decoding implementations to be presented in the next section might provide some ideas on constructing BQ-Trees on GPGPUs by utilizing fine-grained data parallelism.

# 5 GPGPU DECODING: PARALLELIZATION SCHEMES AND IMPLEMENTATION OPTIMIZATIONS

The output of BQ-Tree encoding of large scale raster geospatial data is a collection of BQ-Trees, each represents a compressed bitmap of a bitplane of an M*M ($M=2^m$) raster chunk. The encoded BQ-Trees can be accessed independently with distinct chunk and bitplane combinations to facilitate efficient data processing that requires multiple BQ-Trees. 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. 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 more careful designs. We have designed two parallelization schemes and developed two implementations for each scheme.

## 5.1 Parallelization Scheme 1: Divide Separately and Process (DSP)

The DSP approach is an adoption of the coarse-grained parallelization strategy that has been used for multicore CPUs and GPGPU computing blocks. As illustrated in Fig. 5, given a raster chunk of size M*M ($M=2^m$) and assuming there are $T_n=2^t*2^t$ threads available for a computing block, each thread is responsible for processing a level t subtree and decoding $2^{m-t}*2^{m-t}$ raster cells under the subtree. To make the $T_n$ threads run in parallel, the starting positions of all the t threads in processing the tree node array (for all the m-t tree levels) and the LLQS array are required. These positions can be pre-generated when encoding the raster chunks. Assuming the number of bitplanes is B, the additional storage overheads for the starting positions are $S_n=T_n*(m-t)*B$ integers and $S_l=B*T_n$ integers for the two arrays, respectively. Note that the top t levels of BQ-Tree nodes need to be processed in CPU to make full use of the GPGPU threads.

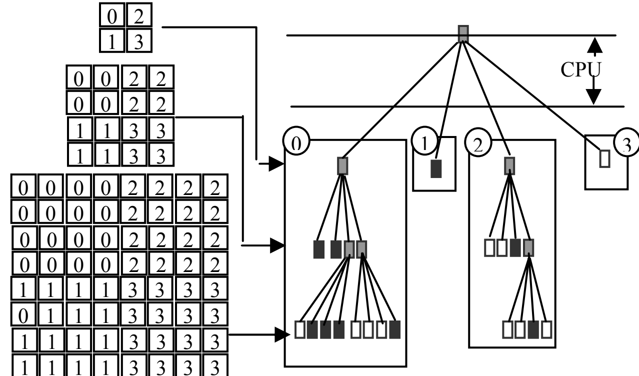


Fig. 5 Illustration of the Divide Separately and Process (DSP) Parallelization Scheme

The major advantage of the DSP algorithm is that it resembles the CPU decoding algorithm closely and it is simple. However, the main disadvantage is that each thread needs to fetch data from non-continuous GPU global memory individually. Except for the first level BQ-tree nodes being processed on GPGPU, the threads access non-continuous global memory locations. GPUs need to issue distinct memory operations for the threads. Unlike CPUs that have large caches (e.g., 256K L2 per-core and 8M L3 shared for Intel Xeon E5520 quad-core CPUs) to significantly reduce memory access costs, GPUs have very limited caches and thus accesses to global memory are more expensive than CPUs. Even the latest Nvidia Fermi GPUs have only a maximum of 48K per-SM L1 cache and 768K L2 cache for all its 14 SMs/448 cores. Given the large number of threads that are launched simultaneously in the decoding kernel (in the order of thousands as detailed in the experiment section), caching in the current generation of GPUs can not help non-coalesced memory

accesses as much as that in CPUs. As shown in the experiment section, DSP based implementations are much less efficient than the PCL based ones (to be detailed in the next subsection). Nevertheless, the parallelization scheme and its implementations can help understand the importance of designing parallel algorithms that fit hardware architectures and serve as baselines for comparisons. We next discuss two DSP implementations.

As discussed in Section 4.2, B bitplane bitmaps need to be accessed simultaneously to decode raster cells. Since the BQ-Trees for the bitplane bitmaps are stored and decoded separately, they need to be combined during the decoding process. There are several possibilities. The first one is, similar to CPU implementation, using global memory as the scratchpad for the combination (as illustrated in Fig. 6). This is straightforward and actually was our first implementation. Unfortunately, it gives us very poor performance. As shown in the experiment section, the performance is so poor that it is only comparable to single thread CPU implementation which makes the GPGPU implementation meaningless in most cases. We found the poor performance was largely due to excessive non-coalesced global memory access. In addition to accessing encoded BQ-Trees and compacted LLQSs in global memory that is unavoidable, the naïve implementation also requires reading and writing bitmaps of all pyramid levels during bitwise composition (combination) processes to restore bitplane bitmaps into the original rasters. Assuming that the raster to be decoded has a size of M*M at each computing block, then the number of reads and writes to global memory by all the threads in the computing block can be roughly estimated as $B*2*(1/3*M^2+M^2)$ where $1/3*M^2$ is for accessing quadtree nodes (also refer to the formula in calculating the size of the pyramid in Section 4.1) and $M^2$ is for accessing LLQSs. We term this implementation as DSP-Naïve.

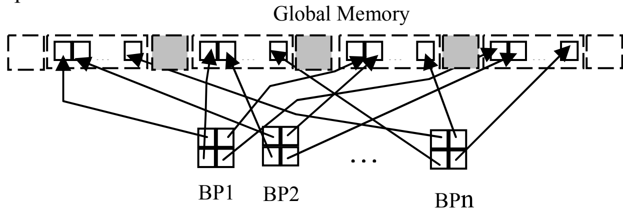


Fig. 6 Illustration of DSP-Naïve Implementation: Excessive Non-Coalesced Global Memory Accesses

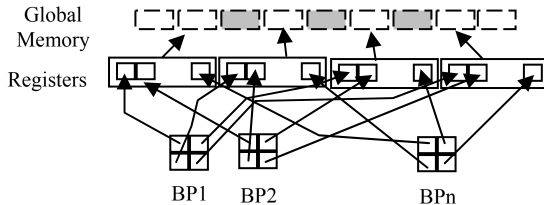


Fig. 7 Illustration of the DSP-Register Implementation: Reducing Excessive Global Memory Accesses by Using Registers

Our optimized DSP implementation, termed as DSP-Optimized, is to utilize GPGPUs' registers to reduce read and write accesses to global device memory. As shown in Fig. 7, four register variables are used per thread when processing a 2*2 quadrant of the BQ-Trees of all B bitplanes. Since all the intermediate results in composting B bitplanes to decode a raster cell use registers and only the register variables need to be written back to global memory, the global memory I/Os are now reduced to 1/B when compared with DSP-Naïve. As shown in the experiment section, DSP-Optimized reduces decoding time to less than 1/3 of that of DSP-Naïve, i.e., a 3X speedup is achieved.

## 5.2 Parallelization Scheme 2: Process Collectively and Loop (PCL)

The PCL scheme adopts a different parallelization strategy than DSP. As shown in Fig. 8, all the threads assigned to a computing block are bundled together to process a quadrant of matrices in a BQ-Tree pyramid during decoding. This collective process is looped over all the quadrants and all levels of the pyramid. Similar to the DSP approach that requires the starting positions of each thread in the tree node array and LLQS array, the PCL approach requires the starting positions in both arrays in order to make the threads assigned to the GPGPU computing blocks work in parallel. However, keeping the positions for all threads is impractical since the storage overhead is overwhelming. A solution is to calculate the positions for all threads on the fly. 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. Clearly the storage overheads for the starting positions (including both 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 $S_l=B*M*M/T_n$. Contrary to the DSP scheme where both $S_n$ and $S_l$ increase as $T_n$ increases, both $S_n$ and $S_l$ decrease as $T_n$ increases (note $T_n=2^t*2^t$) in PCL. This feature makes PCL preferable when the degree of parallelization (in terms of number of threads per computing block) increases. We next present the details of the on-the-fly calculation of the positions in a GPGPU computing block.

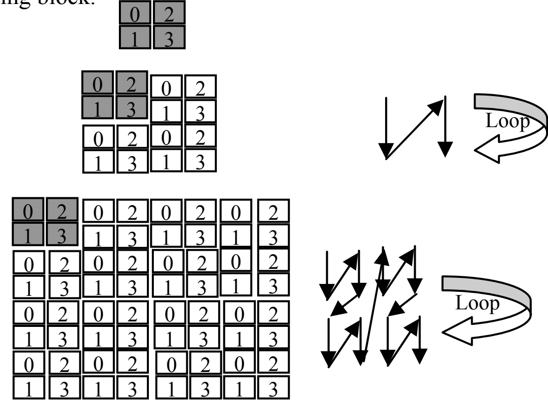


Fig. 8 Illustration of Thread Assignment in the PCL Parallelization Scheme

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. The CUDA code snippet and an example using four threads are shown in Fig. 9. Clearly the time complexity is in the order of $O(\log T_n)$. For $T_n$=256, the process completes in 8 parallel steps. After the starting positions of the threads are computed, they can work independently and achieve good parallel performance. 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.

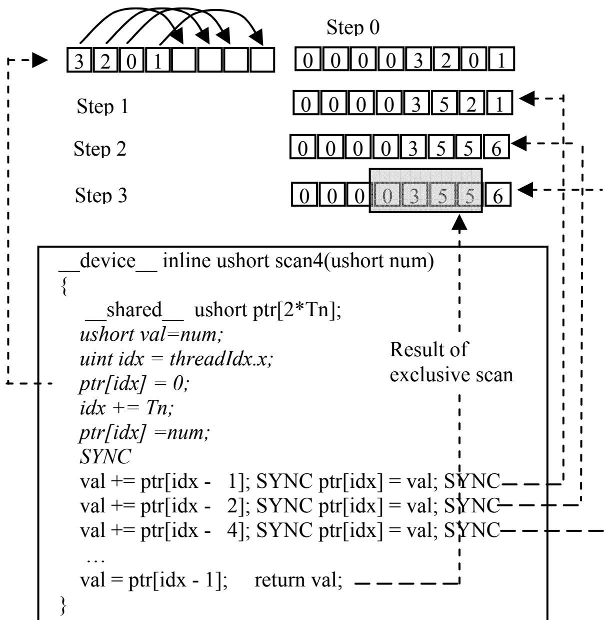


Fig. 9 CUDA Code Snippet and an Example to Illustrate Fast Calculation of Starting Positions using Parallel Scan

Similar to the three DSP implementations, we next present two PCL implementations, namely PCL-Register and PCL-Optimized. Similar to DSP-Optimized, PCL-Register uses GPGPU registers to hold the intermediate results during the bitwise composition process in the last step of decoding. While it is easier to implement, one big problem is that 4*16 registers are needed per thread for this purpose where 4 is because one leaf BQ-Tree node is mapped to four last level quadrants and 16 is due to using a 4*4 quadrant size in forming LLQSs (as short integers). Unfortunately, Nvidia Fermi-based GPUs currently allows 32768 registers shared by 1024 threads in a SM, i.e., 32 registers per thread, when the SMs are fully utilized. Even worse, quite some registers are needed for other parts of the decoding algorithm. While it is possible to use shared memory to alleviate the register stress to a certain extent, we have determined that the PCL-Register implementation is not practical. This motivates us to develop the PCL-Optimized implementation. Before we present the PCL-Optimized implementation in details next, we would like to stress that, thanks to CUDA, the PCL-Register implementation still runs perfectly although spilling register variables to global memory hurts performance significantly. As shown in the experiment section, although the PCL-Register implementation is more than an order better than DSP-Naive, it only achieves about 1/5 performance of PCL-Optimized.

The PCL-Optimized implementation writes individual reconstructed bitmaps to global memory and launches a separate kernel to combine the decoded bitplane bitmaps. 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.

# 6 PERFORMANCE STUDIES

## 6.1 Dataset and Experiment Setup

We use a real NASA MODIS (Moderate Resolution Imaging Spectroradiometer) raster dataset obtained from the Global Land Cover Facility (GLCF) website [30]. 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. 10 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.

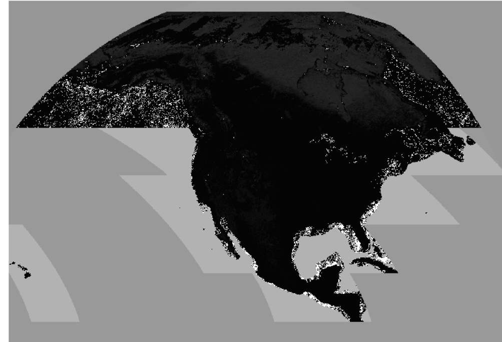

Fig. 10 Grayscale Image of the Experiment Dataset

Our experiments are performed on a SGI Octane III machine. While the machine comes with two identical and independent nodes and four Nvidia Fermi C2050 GPU devices, only one node and a single GPU device on the node is used for experiments. The computing node is equipped with dual Intel Xeon E5520 quadcore CPUs with hyper-threading enabled. As such, the computing node has 8 CPU cores (2.26 GHz) and is capable of launching 16 software threads. The computing node is also equipped with 48 GB 1333MHz DDR3 memory and 4 TB SATA 7200 RPM hard drives. The C2050 GPU card attached to the machine has 448 cores (1.15 GHz). The GPU device also has 3GB GDDR5 graphics memory running at 1.5 GHz clock rate. The GPU device is attached to the motherboard through a PCI-E

x16 slot that can provide a theoretical unidirectional data transmit speed of 4GB/s.

Our primary measurement in this study is the wall-clock running times measured in milliseconds. We do not include data transfer times between CPUs and GPUs in the comparisons among CPU running times and GPGPU running times for the following reasons. First, in many cases, decoded rasters can be used on GPUs for subsequent processing without needing to transfer back to CPUs. Second, as shown later, the CPU to GPU data transfer times, while signficant, is not dominant. Including the data transfer times does not affect speedups dramatically. Third, the encoded data can be pre-loaded to GPUs for time-critical applications such as interactive visual explorations. The processing time is more important than data transfer time in this case. Mixing the two different types of time costs does not reflect the application semantics accurately. Nevertheless, the data transfer times are reported to help understand end-to-end performance.

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. Since encoded byte streams depend on chunk sizes and last level quadrant sizes but not CPU or GPGPU implementations, we report the results in this subsection as tabulated in Table. 1. From Table 1 we can see that, while chunking increases data volumes to be processed, it has minimum effect on both tree node array sizes and LLQS array sizes as the padded raster cells are largely compressed. We can also see that in this particular dataset, using a 4*4 last level quadrant size achieves more than 5% compression ratio than using a 2*2 last level quadrant due to the fact that the quadtree node sizes are significantly decreased while the last level quadrant sizes are only moderately increased. As such, using a 4*4 last level quadrant size provides a better tradeoff.

Table 1 CPU Encoding Results: Sizes (in Bytes) and Compression Ratios (%)

| Chunk Size | 1024*1024 | | 4096*4096 | |
|---|---|---|---|---|
| Data Volume | 771,751,936 | | 805,306,368 | |
| Last Level Quadrant Size | 2*2 | 4*4 | 2*2 | 4*4 |
| Node Array Size | 131,277,689 | 35,700,341 | 131,276,652 | 35,699,304 |
| LLQS Array Size | 136,290,830 | 191,154,696 | 136,290,199 | 191,154,696 |
| Total Encoded Size | 267,568,519 | 226,855,037 | 267,566,851 | 226,854,000 |
| Compression Ratio | 37.88% | 32.12% | 37.88% | 32.12% |

While it is non-trivial to optimize GPGPU implementations even in the cases such as using registers to speed up accesses to global memories (which is the motivation of developing the DSP-Optimized implementation), such optimizations can be easily achieved by setting proper compilation flags on CPUs. We use O3 flag to optimize CPU code for speed that is supported by the gcc compiler on Linux. We report the CPU running times using both non-optimized and optimized code.

## 6.2 CPU Performance

Using OpenMP APIs, we have set the number of current threads to 1, 2, 4, 8 and 16 and measured their running times for both 1024*1024 and 4096*4096 chunk sizes, respectively. The results (both with and without optimization) are tabulated in Table 2. As we can see, the decoding times vary with the following factors: compilation optimization, level of parallelization (number of threads), chunk sizes and last level quadrant sizes. The first two factors, i.e., optimization and multi-threading, play a dominate role while the other two factors are secondary. The running times for "non-optimized+4*4 last level quadrant size" cases are always worse than those of "non-optimized+2*2 last level quadrant size" cases. On the other hand, the "optimized+4*4 last level quadrant size" cases are always better than those of the "optimized+2*2 last level quadrant size" cases. This may suggest that using 4*4 last level quadrant size is more optimization friendly. We can also see that the speedups due to multi-threaded parallelization are almost linear using up to 8 threads for almost all cases regardless of options for optimization, chunk sizes and last level quadrant sizes. The results suggest that the coding framework using BQ-Trees is quite scalable on multicore CPUs. We also notice that using 16 threads only reduces decoding times slightly (3%-13%) and sometimes even performs worse than using 8 threads. This is not surprising since the machine has only 8 physical CPU cores and the decoding algorithms are scalable in utilizing the physical hardware resources. Using more than one thread per CPU core can potentially incur resource contention and reduce performance.

Comparing the best performance of 1,095 milliseconds, which is achieved in the case of 1024 chunk size using 4*4 last level quadrant size and using 8 threads and with optimization, with the worst performance of 55,042 milliseconds, a speedup of more than 50X has been achieved. If we follow the speedup factor chain overlaid with Table 2 (where only one factor is changed per step), we can see that the optimization contributes 6.81X (7462/1095), the parallelization contributes 6.74X (50305/7462) and the last level quadrant size contributes 1.094X (55042/50305).

Table 2 Running times of Decoding on CPUs

| Chunk Size | 1024*1024 No optimization | | 4096*4096 No optimization | | 1024*1024 –O3 | | 4096*4096 -O3 | |
|---|---|---|---|---|---|---|---|---|
| LLQS | 2*2 | 4*4 | 2*2 | 4*4 | 2*2 | 4*4 | 2*2 | 4*4 |
| 1T | 45389 | 50305 | ~~51243~~ | 55042 | 9701 | 7005 | 11329 | 8613 |
| 2T | 23517 | 26512 | 27729 | 31015 | 5116 | 3693 | 6353 | 4815 |
| 4T | 12041 | 13654 | 14402 | 16362 | 2678 | 1917 | 3414 | 2557 |
| 8T | 6908 | 7462 | ~~7728~~ | ~~8835~~ | ~~1571~~ | 1095 | 1889 | 1451 |
| 16T | 5999 | 7054 | 7475 | 8185 | 1554 | 1281 | 1693 | 1487 |

## 6.3 GPGPU Performance

Table 3 lists both the volumes of additional position data that are required to be transferred to GPU device for parallel decoding as well the running times for the four GPGPU implementations, i.e., DSP-Naïve, DSP-Optimized, PCL-Register and PCL-Optimized. We note that the DSP implementations use a 2*2 last level quadrant size while the PCL-implementations use a 4*4 last level quadrant size. We have not tested the other parallelization and last level quadrant size combinations due to time limit. Not shown in Table 3 are the CPU-GPU (host to device) and GPU-CPU (device to host) data transfer times. Depending on the different combinations of chunk sizes, last level quadrant sizes, parallelization schemes and different runs, transferring the encoded data and the auxiliary position data for parallel execution from CPU to GPU takes about 72-77 milliseconds while transferring back the decoded raster from GPU to CPU takes about 215-216 milliseconds. The effective data transfer rates are pretty consistent and are close to the maximum PIC-E *16 limit (4GB/s). Note that we do not include the data transfer times in calculating speedups for the reasons discussed previously.

We have chosen to use 256 threads per computing block, i.e., $T_n$=256, for all the GPGPU implementations to achieve an optimum GPGPU hardware occupancy after considering constraints related to registers and shared memory. Using 256 threads per computing block is also recommended by the GPU vendor and by previous studies. Since each of the 14 SMs can hold up to 4 computing blocks, there are up to 14*4*256=14,336 active threads in the decoding kernels. From Table 3 we can see that, the position data overhead is significant for neither chunk sizes using the PCL parallelization scheme. It is also insignificant for the DSP parallelization scheme using a chunk size of 4096 (1.16%). However, the overhead is significant for the DSP parallelization scheme using a chunk size of 1024 (11.91%). The results reflect different scalabilities of the DSP and the PCL parallelization schemes, as formulated in Section 5. They also suggest that PCL is more preferable to DSP, not only from the running times perspective as detailed next, but also from storage overheads perspective.

Table 3 GPGPU Results: Position Data Overheads and Decoding Running Times

| Chunk Size/Last Level Quadrant Size | 1024*1024 (2*2) | 4096*4096 (4*4) |
|---|---|---|
| DSP Quadtree Position Data (Bytes) | 30,146,560 | 2,752,512 |
| DSP Last Level Position Data (Bytes) | 6,029,312 | 393,216 |
| DSP Position Data Overhead (%) | 11.91% | 1.16% |
| DSP- Naïve Runtime (milliseconds) | 11615 | 9997 |
| DSP- Optimized Runtime (milliseconds) | 3314 | 3037 |
| PCL Quadtree Position Data (Bytes) | 494,592 | 523,776 |
| PCL Last Level Position Data (Bytes) | 1,507,328 | 1,572,864 |
| PCL Position Data Overhead (%) | 0.87% | 0.92% |
| PCL- Register Runtime (milliseconds) | 1053 | 946 |
| PCL – Optimized Runtime (milliseconds) | 190 | 283 |

From Table 3 we can see that, similar to CPU results, the optimized implementations perform much better than non-optimized ones. The DSP-Naïve has a running time of 11615 milliseconds which makes it even more inferior to the optimized single-thread CPU implementation (11329 and 8613 milliseconds using quadrant sizes of 2*2 and 4*4, respectively). Perhaps the most signficant conclusion we can draw from this study, as shown in Table 3 is that the PCL-Optimized implementation achieves a performance of 190 milliseconds, which is a 5.8X speedup (1095/190) when compared with the best CPU implementation on dual quadcore CPUs using 16 threads. This clearly demonstrates the importance of designing a good parallelization scheme to fully utilize parallel hardware capacity on GPGPUs. When compared the best GPGPU performance with the best single-thread CPU performance, we can see a 36.9X speedup (7005/190). Furthermore, when compared the best GPGPU implementation performance with the baseline performance (non-optimized, single-thread CPU implementation), an impressive speedup of 239X to 290X has been observed.

# 7 RELATED WORKS

The research reported in this paper is closely related to our previous work on Binned Min-Max Quadtree (BMMQ-Tree) indexing of large-scale raster geospatial data on GPGPUs [31] which again is a port of the serial design/implementation presented in [32]. This work explores the redundancy of bitplane bitmaps of large-scale raster geospatial data through BQ-Tree coding. The simultaneous lossless compression and indexing makes it suitable for a wider range of applications. Terrain is an important raster geospatial data and there are a few works on visualizing large-scale terrain data using GPGPUs [33]. However, these works are visualization (rather than query) centric and focus on rendering. The rendering window sizes (e.g., 1024*1024) in such applications are much smaller (1-2 orders) than the raster sizes that we are targeting at to answer queries.

The MapReduce framework [34] have attracted signficant interests in parallel processing of large-scale data on shared-nothing clusters [35] and multicore CPUs [29]. In addition, adapting and evolving traditional data management and relational database techniques to multicore CPUs have been research hotspots in recently years [36]. Compression is considered as a viable solution to utilize parallel hardware resources, balance between I/O and computation and speed up query processing on read-only data [37][38]. Although existing research primarily focuses on relational data, we see a similar evolution trend in Spatial Databases and GIS. A few of works experiment on building relational databases on GPGPUs, including indexing [39][40] and compression [41], to speed up query processing. Similar to applying the MapReduce framework to geospatial data [42][43], the techniques currently designed for relational data can potentially be adapted to process large-scale geospatial data.

Our work is also greatly influenced by the works on developing efficient GPGPU primitives, especially the radixsort implementations [44] which motivate us to utilize the parallel scan primitive at the computing block level to compute the positions on the BQ-Tree node array and the LLQS array for all the threads on the fly. The implementation, PCL-Optimized, has turned out to be the most efficient one among all the implementations presented in this study.

# 8 SUMMARY AND CONCLUSIONS

In this study, we have developed the BQ-Tree spatial data structure to code large-scale raster geospatial data. In addition to utilizing the chunk-level coarse grained parallelism on both multicore CPUs and GPGPUs, we have also developed two fine-grained parallelization schemes, namely DSP and PCL, and their four implementations by using different system and application level optimization strategies. Experiments show that the best GPGPU implementation, PCL-Optimized, 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 a 6X speedup than the best dual quadcore CPU implementation using 16 threads and achieves 239-290X speedups than a baseline single thread CPU implementation.

For future work, first of all, as mentioned in the introduction and the motivations sections, we would like to follow the multi-component bitmap indexing framework introduced in [7] to code geospatial rasters that can facilitate exact, range and interval queries, in addition to bitplane level decoding which is the focus of this study. The major differences of the two tasks are only on deriving bitmaps from inputs and compositing bitmaps to derive outputs. The two tasks can share much of the codebase that has been developed in 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 designs and implementations are compared against each other 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 [12] 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.

# 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. Shusterman, E. and M. Feder (1994). Image Compression Via Improved Quadtree Decomposition Algorithms. IEEE Transactions on Image Processing **3**(2): 207-215.
6. Senbel, S. and H. Abdel-Wahab (1997). A Quadtree-based Image Encoding Scheme for Real-time Communication. IEEE Conference on Multimedia Computing and Systems, 143-150.
7. 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.
8. Hennessy, J. L. and D. A. Patterson (2006). Computer Architecture: A Quantitative Approach (4th ed.). Morgan Kaufmann.
9. Asanovic, K., R. Bodik, et al. (2009). A view of the parallel computing landscape. J Commun. ACM **52**(10): 56-67.
10. Stockinger, K., J. Shalf, et al. (2005). Query-Driven Visualization of Large Data Sets. IEEE Visualization Conference: 167-174.
11. Samet, H. (2004). Object-based and image-based object representations. ACM Computing Surveys **36:** 159-217.
12. SciDB. http://www.scidb.org/.
13. Chan, C.-Y. and Y. E. Ioannidis (1998). Bitmap index design and evaluation. ACM SIGMOD conference, 355-366.
14. Chan, C.-Y. and Y. E. Ioannidis (1999). An efficient bitmap encoding scheme for selection queries. ACM SIGMOD conference 215-226.
15. Wu, K., W. Koegler, et al. (2003). Using bitmap index for interactive exploration of large datasets. SSDBM Conference 65-74.
16. Wu, K. S., E. J. Otoo, et al. (2006). Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems **31**(1): 1-38.
17. Sinha, R. R. and M. Winslett (2007). Multi-resolution bitmap indexes for scientific data, ACM Trans. Database Syst. 32**:** 16.
18. Lemire, D., O. Kaser, et al. (2010). Sorting improves word-aligned bitmap indexes. Data & Knowledge Engineering 69(1): 3-28.
19. FastBit. https://sdm.lbl.gov/fastbit/.
20. Shekhar, S. and S. Chawla (2003). Spatial Databases: A Tour. Prentice Hall.
21. Owens, J. D., D. Luebke, et al. (2007). A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1): 80-113.
22. Hwu, W.-M. W. (2011). GPU Computing Gems: Emerald Edition. Morgan Kaufmann.
23. Nvidia. Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_home_new.html
24. Owens, J. D., M. Houston, et al. (2008). GPU computing. Proceedings of the IEEE **96**(5): 879-899.
25. Garland, M. and D. B. Kirk (2010). Understanding throughput-oriented architectures. J Commun. ACM **53**(11): 58-66.
26. Rao, J. and K. A. Ross (1999). Cache Conscious Indexing for Decision-Support in Main Memory. VLDB conference, 78-89.
27. Rao, J. and K. A. Ross (2000). Making B+- trees cache conscious in main memory. SIGMOD 2000 Conference: 475-486.
28. Hamming distance. http://en.wikipedia.org/wiki/Hamming_distance.
29. Yoo, R. M., A. Romano, et al. (2009). Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. IEEE Symposium on Workload Characterization, 198-207.
30. Global Land Cover Facility (GLCF) MODIS 500m North America Dataset, ftp://ftp.glcf.umiacs.umd.edu/modis/500m/North_America/
31. Zhang, J., S. You, et al. (2010). Indexing large-scale raster geospatial data using massively parallel GPGPU computing. ACM-GIS Conference.
32. Zhang, J. and S. You (2010). Supporting Web-based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree. SSDBM Conference, 379-396.
33. Amara, Y. and X. Marsault (2009). A GPU Tile-Load-Map architecture for terrain rendering: theory and applications. Visual Computer **25**(8): 805-824.
34. Dean, J. and S. Ghemawat (2008). MapReduce: simplified data processing on large clusters. J Commun. ACM **51**(1): 107-113.
35. Pavlo, A., E. Paulson, et al. (2009). A comparison of approaches to large-scale data analysis. ACM SIGMOD conference. 165-178.
36. Cieslewicz, J. and K. A. Ross (2008). Database optimizations for modern hardware. Proceedings of the IEEE **96**(5): 863-878.
37. Abadi, D., S. Madden, et al. (2006). Integrating compression and execution in column-oriented database systems. ACM SIGMOD conference, 671-682.
38. Holloway, A. L., V. Raman, et al. (2007). How to barter bits for chronons: compression and bandwidth trade offs for database scans. ACM SIGMOD Conference, 389-400.
39. He, B. S., M. Lu, et al. (2009). Relational Query Coprocessing on Graphics Processors. ACM Transactions on Database Systems **34**(4).
40. Bakkum, P. and K. Skadron (2010). Accelerating SQL database operations on a GPU with CUDA. GPGPU '10 workshop, 94-103.
41. Fang, W., B. He, et al. (2010). Database compression on graphics processors, VLDB Endowment. **3:** 670-680.
42. Cary, A., Z. Sun, et al. (2009). Experiences on Processing Spatial Data with MapReduce. SSDBM Conference, 302-319.
43. Akdogan, A., U. Demiryurek, et al. (2010). Voronoi-Based Geospatial Query Processing with MapReduce. IEEE CloudCom Conference. 9-16.
44. Satish, N., M. Harris, et al. (2009). Designing efficient sorting algorithms for manycore GPUs. IEEE Symposium on Parallel\&Distributed Processing, 1-10.