Parallel QuadTree Encoding of Large-Scale Raster Geospatial Data on Multicore CPUs and GPGPUs

Nathalie Kaligirwa¹, Eleazar Leal¹, Le Gruenwald¹, Jianting Zhang² and Simin You³

¹School of Computer Science, University of Oklahoma, Norman, OK 73071, USA
{j Nathalie.Kaligirwa-1, eleal, ggruenwald}@ou.edu
²Dept. of Computer Science, City College of New York, New York City, NY 10031, USA
jzhang@cs.ccny.cuny.edu
³Dept. of Computer Science, CUNY Graduate Center, New York City, NY 10031 USA
syou@gc.cuny.edu

ABSTRACT
Global remote sensing and large-scale environment modeling have generated vast amounts of raster geospatial images. To gain a better understanding of this data, researchers are interested in performing spatial queries over them, and the computation of those queries’ results is greatly facilitated by the existence of spatial indices. Additionally, though there have been major advances in computational power, the I/O transfer is becoming the major bottleneck in the overall system performance. One of the solutions to the I/O channel bandwidth issue is to compress data first and then send it over an I/O channel. Therefore, a compression technique that achieves not only indexing but also compression is highly desirable. The performance of geospatial raster data compression and indexing can benefit from high performance technologies such as General Purpose Graphics Processing Units (GPGPUs) that are increasingly becoming affordable. Our purpose in this article is two-fold: we will first present issues relating to compressing geospatial raster data and popular compression techniques. Afterwards, we will present a parallel implementation of the compression of geospatial raster data using a cache-conscious quadtree data structure. Experiments show that our GPGPU implementation is capable of constructing a BQ-Tree encoding for the 16-bit NASA MODIS geospatial raster of up to 950 MB in less than one second with an Nvidia C2050 GPU card. This performance represents a 3x speedup with respect to our best implementation of the same algorithm on multi-core CPUs with 16 threads, and up to a 12x speedup compared with a multi-core implementation (with 16 threads) of the popular Zlib compression library.

Categories and Subject Descriptors
H.2.8 [Database Management]: Database applications – Spatial Databases and GIS

General Terms
Measurement, Performance, Experimentation, Algorithm.

Keywords
High Performance, GPGPU, GIS Data, Compression.

1. INTRODUCTION
Geospatial data refers to data that is defined spatially by four dimensions: three of them corresponds to x, y and z coordinates, and the fourth one is time. Geospatial data can be represented as images; images can be represented in two major ways: either by using a vector representation, in which the elements to be displayed are represented by specifying collections of vertices, along with collections of geometric primitives such as lines, Bezier curves, etc., or by using a raster representation, in which the elements to be displayed are represented through a collection of pixels. Many Geographical Information Systems (GIS) work on large geospatial datasets in raster format. Indeed, raster images are more suitable to represent complex images with non-uniform colors and shading [1]. With the rapid development of information technology, remote sensing imagery, GPS technologies and so on, there is a huge amount of large-scale raster geospatial data available. While high performance modern CPUs can perform up to 500 single precision giga-floating points operations per second, the I/O speed is limited to around 50-gigabytes per second [2]. This makes I/O transfer of data one of the main bottlenecks in processing large-scale geospatial data.

One way of reducing the I/O transfer time is to compress the data so it occupies less storage and lowers disk I/O and data transfer time. Data compression on images can be either lossy(the compressed data is only an approximation of the original data) or lossless (the original data can be totally reconstructed from the compressed data); and the latter is desirable for GIS applications that need to maintain data quality in order to perform analyses such as spatial queries. With the large-scale data available today, there is a need to explore high performance computing in an effort to boost the performance of applications. CPU processors have hit a performance stall due to the power limitations on semi-conductors [3]. Therefore, multi-core CPUs and GPGPUs have distinguished themselves as the two leading hardware architectures for boosting performance due to their parallel nature. GPGPUs (GPUs) are co-processors focused on maximizing throughput —number of instructions executed per unit of time—, rather than minimizing latency —the time it takes to execute a single instruction. This approach meant that on-chip space that would normally be dedicated to minimizing latency, such as branch predictors, out-of-order execution, and large caches, could instead be used to accommodate more functional units [4].
The low cost of GPGPUs—both overall device cost and energetic cost per instruction (GPGPUs have higher performance per watt than CPUs [5]) and their high availability, and the their throughput that is an order of magnitude greater than commercially available multicore chips [4]—caught the attention of researchers from many different fields [6]. Nvidia (a GPU vendor), designed a C-like language CUDA (Computing Unified Device Architecture) to facilitate the implementation of general-purpose algorithms on GPGPUs. GPGPUs that are CUDA-capable are conventionally called General Purpose Graphical Processing Units (GPGPU). Recent research in high performance computing for databases has focused on improving query processing instead of compression as a strategy to reduce the I/O cost [7][8][9]. Parallel processing technologies have also been used to increase the disk/output bandwidth for spatial data management systems by using either multi-core CPUs [10], by the increasingly popular Google MapReduce platform[11], or GPGPU technologies. A number of reports such as [5], [12] and [13] have provided much needed insight in how to implement general-purpose algorithms on GPGPUs. There is still a huge gap in the literature available for processing large-scale geographical data in general, and raster geospatial data specifically.

In this paper we present a method for compressing raster geospatial data using GPGPU technologies. In the next sections, we first discuss popular compression techniques (Section 2), then we introduce the data structure used for compression in Section 3, Section 4 gives an overview of GPGPGU technologies, and Section 5 discusses the GPGPU implementation proposed. We conclude with a performance evaluation in Section 6.

2. COMPRESSION RESEARCH ISSUES AND RELATED WORK

In this section we first classify popular techniques according to the common issues to be taken into consideration for the compression of raster geospatial data. This classification provides a useful perspective on which we based our choice of using the BQ-Tree data structure. Data compression has been an active field ever since the need to transfer data over a network as well as reducing storage on disk or CPU main memory have been a necessity. Gradually, compression methods specifically targeted for images were developed using inherent properties of those images[16]. Image compression techniques have been extensively used on geospatial data [14]; however because GIS applications are interested in drawing useful information from data, state-of-the-art compression methods for geospatial data have been able to facilitate further analyses. As a result, for geospatial data compression, we are concerned with balancing many parameters such as compression ratio, compression speed, fast spatial access, and spatial locality to not only reduce storage needs but also to improve the overall performance of further analyses on the data. Existing compression techniques can be classified in these three categories:

A. Compression-Ratio Oriented Compression Techniques:
Entropy encoding techniques, which are implemented independently of the type of medium to be compressed, can be applied to raster images [15]. Methods include Huffman Coding, Shannon- Fano Coding, and arithmetic coding[16] have been widely used in raster images compression. Nevertheless, the spatial locality of the data is not preserved because these techniques work on one-dimensional data and do not use the semantics of images as a basis for compression. Popular semantic-dependent techniques (techniques that take into account the layout of the data), which have been studied ([17][18][19]), such as run-length encoding, block encoding and difference mapping, rely on the premise that most images have high local redundancy [17]. However, these methods typically process the raster image in row, column, or zig-zag order, thus losing the spatial order of the data.

2 Compression-Speed Oriented Compression Techniques:
FPC, a technique developed in [21] losslessly encodes double precision floating-point data by predicting values and storing the difference between the actual values and the predicted values. It reduces compression time by using fast integer operations. Although, this technique has the advantage of being compression-time aware, it does fall short on solving the issues of fast spatial access and spatial locality. Another data compression approach introduced in [22], called the ISOBAR pre-conditioner, divides the data into chunks of bytes and then decides which of these would achieve a good compression ratio (relatively uniform data) and it also selects which compression method to use (either zlib [41] or bzip2). This method achieves a higher compression speed compared to standard techniques such as zlib; however, if applied to raster geospatial data, although it allows for fast spatial access at the byte-level, it does not preserve spatial locality because it analyzes and compresses the data in row or column-order only. Compressed bitmap indexes techniques such as the Word-Aligned Hybrid code [20] do not directly increase the compression speed, but because they allow direct queries on compressed data, decompression time is minimized; indeed in some cases, a good compression technique has to support fast decompression. These techniques are not only concerned with achieving a decent compression ratio, but also with reducing the time it takes for direct queries on compressed data, even if it means achieving lower compression rates. The Byte-Aligned Bitmap Code (BBC) is another specialized method that increases compression ratio while maintaining fast query processing times on compressed data[20].

C. Fast Spatial Access and Spatial Locality Oriented Compression Techniques:
Multi-dimensional data structures fall within this category when applied to spatial compression of raster images. Popular data structures include the R-tree[40], R*-Tree[39], and B-Tree[41] and the family of quadtrees [23]. One of the most popular data structures, the R-Tree, has been extensively used for indexing spatial data. Works such as [24] and [25] have proposed techniques for reducing the size of R-trees thus effectively making them ideal for data compression, fast spatial access and maintaining spatial locality. Quadtree is another popular indexing structure because of its simplicity in indexing, compressing and querying [1]. A recent technique called HFrC [26] was developed to encode height field data by using Bezier surfaces and control points to compress the data. The compressed data is stored across three layers. This technique effectively allows fast access because each height field value can be independently computed using the Bezier surface control points. Additionally, the compressed data is stored in layers which can be accessed independently, even though it is at the expense of accuracy. However this technique works best for data representing smooth surfaces or with high continuity properties, which cannot be guaranteed for raster geospatial data.

From the compression techniques discussed, we chose to use the quadtree data structure because of a number of reasons. Firstly,
its hierarchical structure maintains spatial locality and fast spatial access. Secondly, if implemented as a cache-conscious data structure, it can greatly reduce the I/O bandwidth [27] thus increasing the overall compression speed. Additionally, a cache-conscious implementation of the quadtree further reduces the storage space by removing the need for storing pointers (refer to Section 3). In this work, we will use a variant of the quadtree called the BQ-Tree [27] to compress raster geospatial data. The BQ-Tree not only handles well all the major issues of compressing raster geospatial data but its highly parallelizable nature offers opportunities for using the huge processing power of GPGPUs technologies. A number of popular image compression techniques have been implemented on GPGPUs. In works such as [28] and [29], transform methods that act as a pre-conditioning step for compression have been optimized using GPGPUs. However, these techniques do not facilitate query processing and some of them such as the one presented in [29] are lossy.

A few bitmap compression techniques have also been implemented on GPGPUs. For example, GPU-WAH [31] is a parallelized version of the popular WAH [20]. [30] has proposed a parallelized version of Run-Length Encoding and the CULZSS dictionary-based compression [19] is a parallelized version of the L277 compression technique implemented on GPGPUs. A quadtree hybrid implementation on CPU-GPGPUs is proposed by [42] but has the disadvantage of being a pointer-based implementation, which would incur a storage penalty if used on large-scale data (refer to Section 3). Our goal is to present an efficient implementation of the BQ-Tree on GPGPUs for the compression of large-scale raster geospatial data.

3. BQ-TREE DATA STRUCTURE

Given a finite set of points in a 2-dimensional space, the quadtree datastructure partitions the space in such a way that allows for efficient retrieval. A quadtree is one of the best-known spatial data-structures and is also a particular kind of point-access-method (PAM) [1] There are multiple kinds of quadtrees [33], we focus on region quadtrees which can be used to represent binary images, multicolored data, or grayscale images [27].

Given a bitplane bitmap of a raster R of size \(2^m \times 2^m\), the bitmap will be represented as a tree with a branching factor of four, where black leaf nodes represents regions of the raster (referred to as quadquadrants) composed of only 0s, white leaf nodes represent quadrants composed of only 1s, and gray nodes represent internal nodes. The classic quadtree is a pointer based implementation where pointers represent the address of tree nodes. On modern architectures, a pointer will occupy at least 4 bytes of memory on 32-bit machines and 8 bytes on 64-bits machines, which is unacceptable for bitplane bitmap compression, where a leaf node requires only 2 bits to store the value of quadrants. The BQ-Tree data structure proposed in [27] provides a solution to this issue by using a cache-conscious data structure. The BQ-Tree is stored as a one-dimensional array, where quadrants are stored contiguously following a recursive Z-order[24]. The BQ-Tree eliminates the need to use pointers; with only the position of the first node of a level l of a tree, one can easily access any other node on l.

Secondly, building a quadtree requires examining every pixel of the image and then recursively generating higher levels of the tree by visiting every lower level node. As the size of available raster geospatial data increases, the available CPU processing power might not be enough to efficiently compress the data. Indeed, although modern CPU processors can perform billions of single-precision floating-point operations per second [2], there has been a stall in processing power. This has prompted us to use GPGPU technologies to increase the performance of the BQ-Tree in order to achieve the highest possible compression speed on large-scale geospatial data.

3.1 BQ-Tree Description

Assuming we are given a bitmap \(B\) of size \(2^m \times 2^m\) and depth D (the maximum number of bitplanes for each bitmap cell), a last-level quadrant of size \(2^q \times 2^q\) is chosen such that \(q < m\). BQtrees representing the bitmap \(M\) are D pyramids that have exactly \(m - q + 1\) levels, where the top level (containing only the root) is level number 0 and the bottom level is level number \(m - q + 1\). Given levels \(0,1,...,i,...,m-q\), the \(i\)th level in the pyramid (or tree) contains \(2^i \times 2^i\) elements.

For each of the D bitplanes, the bitmap can be logically subdivided into last-level quadrants which as we have mentioned have size \(2^q \times 2^q\) (see Figure 1 where 2 different possible subdivisions are shown: on the left we use \(q = 1\) and on the right we use \(q = 2\)). Given that the bottom-most level contains \(2^m-q \times 2^m-q\) elements, there is a one-to-one correspondence between the last-level quadrants and the nodes in the last-level shown in Figure 2. In this figure, we can see that in the matrix of level 2 the entry located at position (0,1) is 01 because its corresponding last-level quadrant (highlighted within a circle in the last-level) contains both 0s and 1s.

A node in the last-level of the BQ-Tree will contain the code 00 if all the \(2^q \times 2^q\) elements of its corresponding last-level quadrant are 0s, 11 if all the elements of this last-level quadrant are 1s, and 01 if the last-level quadrant contains both 0s and 1s. See Figure 2 where the last-level quadrants have size \(2^q \times 2^q\) \((q = 1)\). In this figure, the matrix corresponding to level 2 entry (0,0) contains the code 11 because its corresponding last-level quadrant entries (at positions (0,0),(0,1),(1,0),(1,1) in level 3) all contain a 1.

![Figure 1: Choice of size for the Last-Level Quadrants (LLQs) of a bitplane. On the left we can see sixteen LLQs if we choose a last-level quadrant size of \(2 \times 2\) \((q = 1)\). If we instead choose \(4 \times 4\) \((q = 2)\) last-level quadrants, we would have the division shown on the right of the figure with only 4 LLQs.](image-url)
For any given node $n$ at level $i$ (but not the last-level), it will contain the value 00 or 11 if its four children all have codes 00 or 11, respectively. In the case where there are at least two children of $n$ with different codes, then the code for $n$ is 01. For example, in Figure 2 we can see that the entry $(1,1)$ of the level 1 matrix is 00 (marked with a striped circle) because its corresponding four children (entries $(2,2),(2,3),(3,2),(3,3)$, marked with a striped circle in level 2) are all 00.

### 3.2 Encoding

In this subsection, we explain how BQ-Trees are physically encoded. The fundamental idea behind a BQ-Tree is that all its nodes are linearized, that is, for any non-leaf node, the encodings for its four children are placed in adjacent positions in memory. Another fundamental idea is that we use a variation of a dictionary encoding [37]: any node $n$ that is not a last-level node of the pyramid whose children are all 0s (or are all 1s) will be encoded with only 2 bits: 00 (or 11 respectively).

Given a bitplane of size $2^m \times 2^m$ and a last-level quadrant of size $2^q \times 2^q$, its physical encoding as a BQ-Tree consists of 2 arrays: the Pyramid Array and the Last-Level Quadrant Signatures (LLQS) Array. The physical encoding of a BQ-Tree is illustrated in Figure 3, which shows the construction of the pyramid array, and Figure 4, which show the construction of the LLQS array. In the following sections, we describe how these two arrays are constructed.

The pyramid array of a BQ-Tree consists of a linearization of each of the levels of the pyramid starting from level 1 (level 0 with the root as its only element is already linearized; so we ignore it) down to level $m - q$. Starting from level 1, we concatenate the codes for its four nodes following the Z-order. As we can see in Figure 3, if we perform a Z-order traversal of level 1, we would visit the entries of this level in the following order: $(0,0), (1,0), (0,1)$ and $(0,0)$. As we do this, we write the values of those entries to the pyramid array. So this is why in Figure 3, the pyramid array starts with 01 11, 01, 00, which constitute the linearization of level 1. After this, we perform a traversal of level 2, which produces the entries 11, 11, 01, 01, 11, 11, 11, 00, 01, 01, 00, 00, 00, 00, which are concatenated to the entries of level 1. We proceed analogously for the following levels: for level $i$, we concatenate the codes for its $2^{i-1} \times 2^{i-1}$ nodes in Z-order, and append them to the linearization of level $i - 1$ (see Figure 3 inside the Pyramid Array box where the sequence of bits for level 2 follows the sequence of bits for level 1).

Now, in the above linearization there is an opportunity to prune redundant nodes of the BQ-Tree. When performing the linearization for any of these levels, through the use of the Z-order ordering (which is a curve that establishes a linear ordering over the Cartesian plane such that if two points are close in this plane, then they are close to each other in the curve [1]), we can group the codes corresponding to the $2^{i-1} \times 2^{i-1}$ nodes in any given level $i$ into groups of 4 codes of 2 bits each. For example, in Figure 3 inside the pyramid array, we can make a group of 4 entries at a time. The first group consists of 00, 11 and 00. The second group consists of 11, 11, 01, and so on. All the nodes whose codes fall within the same group $G$ of 4 entries will all be children of the same node. For example, in the pyramid array of Figure 3 we see that the second group of 4 entries (which is composed of 11, 11, 01, 01) comes from the entries $(0,0), (1,0), (0,1)$ and $(1,1)$ of the matrix corresponding to level 2, and they all share the same parent node which is the entry $(0,0)$ of the matrix corresponding to level 1. Hence, if the parent node $p$ of group $G$ has code 00 or 11 then there is no need to output any of the codes corresponding to group $G$. This is simply because the parent $p$ already encodes the values for all its children. An example of this can be seen in Figure 3 inside the Pyramid Array box, where there are two subsequences of bits that are shaded and have been stricken out. This is because in level 1 there is an entry 11 — the bottom left — whose four children are also 11, and here lies the redundancy: there is no need to store the four children with 11s since the parent already contains this information. The same observation applies to the bottom-right entry in level 1 which has a value 00: all its children are 0, so there is no need to store the code for its children.

### 3.2.2 Construction of the LLQS Array

To construct the LLQS array, we visit all quadrants in Z-order, with the idea that entries that are near in the bitmap will also end up close to each other in the array. Every time we enter a quadrant, we must output its $2^q \times 2^q$ elements in row-major order to the LLQS array. This procedure is illustrated in Figure 4. In this figure, each last-level quadrant has size $2 \times 2$ and is visited in Z-order. So in Figure 4, the first quadrant that is visited is the one shaded gray in the upper left corner containing only 1s. We then write its four 1s to the LLQS array in row-major order. This is why in the lower part of that figure we see that the left-most part of the pyramid array is marked ‘Quad. 1’.
Now the second quadrant visited in Z-order is the one just below the first one with no shading and with all its entries equal to 1s. After concatenating its four 1s in row-major order to the pyramid array we can see those four 1s identified in the figure as ‘Quad. 2’. We proceed analogously for the remaining 14 last-level quadrants.

Again, there is an opportunity to prune the information from the LLQS array. Every time we encounter a uniform last-level quadrant (having either all entries with 0s or all with 1s), we can discard it. The reason we can prune these entries is because level \( m - q \) (in this case \( m - q = 3 - 1 = 2 \)) will indicate if the corresponding last-level quadrant is uniform or not. This is shown inside the box “Last-Level Quadrant (2×2) Signatures (LLQS) Array” in Figure 4 where there are shaded entries that have been stricken out.

![Figure 4: Construction of the LLQS Array](image)

**3.3. BQ-Tree encoding is lossless**

In this subsection we prove that BQ-Tree encoding is lossless, that is, given the pyramid and the last-level quadrant arrays, we can reconstruct the original raster without losing information. To do that, we will show that each step of our algorithm can be reversed. For this, we will prove that given the compressed pyramid and compressed last-level quadrant arrays, we can obtain the uncompressed pyramid array, and then we can obtain the uncompressed last-level quadrant array.

Given the uncompressed last-level quadrant, it is immediate to see (from Figure 2 for example) that we can get the original raster by taking the single-bit entry at position \((i,j)\) of bitplane \( k \) to be the \( k^{th} \) bit of the entry at position \((i,j)\) of the raster bitmap.

**Lemma 4.3.1:** The compression of the pyramid array can be reversed.

By induction on the number \( k \) of half-nybbles (a nybble consists of 2 bits) with values 00 or 11 that remain in the compressed pyramid in levels 1 through \( m - q - 1 \), such that position \( 4(i + 1) \) of the compressed pyramid is not 0000000 or 1111111.

**Base Case:** \( k = 0 \). Since there are no half-nybbles with values 00 or 11 in levels 1 through \( m - q - 1 \), the pyramid array compression stage did not change the original pyramid array, so the compression of the pyramid array can be reversed.

**Inductive Case:** Assume the lemma to be true for some \( k = n \geq 1 \), let us see that it holds for \( k = n + 1 \). Let \( i \) be the position of the first half-nybble with value 00 or 11 (shown in Figure 5 marked with a circle). Since this is the case, we know that at position \( 4(i + 1) \) (shown in Figure 5, pointed to by an arrow) there were 4 half-nybbles (corresponding to the children of the node at \( i \)) with values 00 (or 11) that were removed. So to invert the compression of the pyramid array we just add 4 half-nybbles at position \( 4(i + 1) \). Now, the pyramid array will have \( n \) half-nybbles with values 00 or 11 in levels 1 through \( m - q - 1 \). By our inductive hypothesis, the lemma holds and we can reverse the compression of the pyramid array.

![Figure 5: Illustration of the proof that the pyramid compression is reversible](image)

**Lemma 4.3.2:** The compression of the last-level quadrant array can be reversed.

By induction on the number \( k \) of half-nybbles (a nybble consists of 2 bits) with values 00 or 11 that remain in the uncompressed pyramid in level \( m - q \), such that position \( 4i \) of the compressed last-level quadrant array is not 0000 or 1111.

**Base Case:** \( k = 0 \). Since there are no half-nybbles with values 00 or 11 in level \( m - q \) in the uncompressed pyramid, then the last-level quadrant array was not altered during the compression stage. This is because all last-level quadrants were non-homogeneous and the compression only removes homogeneous last-level quadrants.

**Inductive Case:** Assume the lemma to be true for some \( k = n \geq 1 \), let us see that it holds for \( k = n + 1 \). Let \( i \) be the position of the first half-nybble with value 00 or 11 in level \( m - q \). Given that this is the case, we know that at position \( 4i \) in the compressed last-level quadrant array we need to insert either 0000 or 1111 depending on whether the half-nybble located at position \( i \) is 00 or 11 respectively. Since there remain \( n \) half-nybbles such that position \( 4i \) of the compressed last-level quadrant array is not 0000 or 1111, by inductive hypothesis our lemma holds.

### 4. GPGPU PROGRAMMING MODEL AND HARDWARE IMPLEMENTATION

#### 4.1 GPGPU Programming Model: CUDA

**4.1.1 Kernels**

The Computing Unified Device Architecture (CUDA) is a programming interface that supports the execution of general-
purpose programs written in C, C++, Fortran, and other program on NVIDIA GPGPUs. Typically CUDA contains two types of code: code that runs on the CPU processor (host) and code that runs on the GPGPU processors (device) called kernels. Host CUDA code is compiled using standard compilers while the device code (kernels) is converted into a GPGPU intermediate language called PTX, which later is translated into binary code that is optimized to run on GPGPUs [34].

4.1.2. Thread Hierarchy in CUDA
CUDA threads are logically arranged in a 2D grid that is itself subdivided into uniform 3D blocks, and blocks are grouped to form a grid. This hierarchy is very important because it defines how memory is assigned to different threads in a grid. All the threads in a thread block execute concurrently and communicate amongst themselves through shared memory. Each thread has a unique ID that can be used to assign to it a particular task. All blocks within a grid execute the same kernel, and they are in charge of writing data from and to global memory. Thread blocks also have unique IDs that identify them at the grid level.

4.1.3 GPGPU Memory Hierarchy
GPGPU memory hierarchy is built following the threads hierarchy described in the previous section. As a rule of thumb, the more threads access a memory space, the slower it is to fetch data from it. In a GPGPU there are 5 memory spaces that the programmer must take into consideration [4][34]: The register file contains all the processor registers used by a thread. The global device memory is shared among all the blocks of a kernel. This type of memory is called global, because all the threads can access it. Global device memory has an access latency of a 100 cycles on average. The local memory, which is allocated for each thread, takes a 100 cycles on average to access. There is also the shared memory, which can only be accessed by all the threads within in a thread block. Accessing shared memory takes 4-32 cycles on average. Finally, texture and constant memories are read-only memory spaces that are similar to global memory in the sense that all the threads have access to them. Both these memories are read-only from the device and only the CPU can write data in these memories. If texture and constant memories are cached, only 4 cycles are required to fetch data; otherwise, it takes 100 cycles.

4.2 GPGPU Hardware Implementation
In our experiment we used a SGI Octane III machine which is equipped with two Nvidia Fermi C2050 cards, of which we only use one. The Fermi architecture compared with previous architectures, increases double precision operation performance, and introduces true cache hierarchy as well as more shared memory. Our GPGPU card contains 14 stream multi-processors (SMs) each of which has 32 processing cores. Each core of every SM (stream multi-processor) can perform floating-point and integer operations, and has up to 64K of local RAM that can be partitioned into cache and shared memory. Each processing core can launch up to 1536 threads. The GPGPU device connects with the CPU using a PCI-Express bus. The Fermi architecture supports up to 6GB of GDDR5 DRAM memory.

From the hardware specification given above and the threads hierarchy described in subsection 4.1.2, a hardware hierarchy is as follows: each GPU executes one or more kernel grids; each SM executes one or more thread blocks; and the processing cores in the SM execute threads. The number of thread blocks to be allocated to each SM is decided by the GigaThread global scheduler based on the number of thread blocks and the number of threads per block. During kernel execution, threads are grouped in warps of 32 threads; and all threads in one warp execute one instruction at a time.

5. BQ-Tree GPGPU IMPLEMENTATION
Given that the decoding of BQ-Trees may need to be performed multiple times during the execution of queries, and that the encoding of BQ-Trees is performed at most once, we chose to encode BQ-Trees using a Last-Level Quadrant size of 4x4, which offers, for the decoding of the NASA Modis North America raster that is used, a better compression ratio than the 2x2 LLQ size [27].

The characteristics of this dataset are explained in Section 6. We have two implementations, the multi-block-per-tile and the one-block-per-tile. For both implementations, the whole raster is first split into uniform tiles of size 1024x1024. For the multi-block-per-tile implementation, we use simultaneously all the SMs of the GPGPU card to encode each tile. Then, by iterating over all the tiles, we encode the whole raster. In the second implementation, the one-block-per-tile, a number of thread blocks equal to the number of tiles is launched in parallel, and each thread block works on one tile. As illustrated in Figure 6 below, the BQ-Tree encoding algorithm for GPGPUs is implemented in several kernels.

Except for the last step which is the compression of the LLQS array and the pyramid array, both implementations have the same kernels and they only differ on how data is allocated to thread blocks. The first kernel decomposes the whole bitmap into different biplanes arrays, the second kernel calculates the last-level quadrant array (LLQS), the third and fourth kernel builds the pyramid, and the rest of the kernels compress the LLQS array and the BQ-Tree.

**Kernel 1- Decomposition of bitmap into biplanes**: The first kernel reads the raster data (made of cells of 16 bits) and splits it into 16 1-bit (bitplane) rasters.

**Multi-block-per-tile approach**: To achieve this, we use thread blocks of size 16x16 with each thread reading 16 pixel values. The threads will then compact each ith bit of every one of the 16 pixels in a new ith short. After the 16 shorts are generated, they are then sent to global memory, with each ith short being sent to the ith biplane array.

**One-block-per-tile approach**: The process is the same as the one described above but each thread block works on a different tile and a number of thread blocks equal to the number of tiles is launched.

**Kernel 2- LLQS Array Generation**: The second kernel reads each of the 1-bit biplane arrays, and outputs last-level quadrant arrays (LLQS) with a 4x4 last-level quadrant.

**Multi-block-per-tile approach**: For this kernel, 16 thread blocks (each taking care of one biplane raster of the whole raster) of 16x16 threads were used; and each thread reads 16 consecutive elements column-wise. This ensures that each threads reads exactly a chunk of 16x16 bits. The chunk is then subdivided into 4x4 quadrants in z-order. The z-order calculated for each 4x4 last-level quadrants is mapped into the LLQS array by using the thread id and the thread block id to correctly place it in the LLQS array.

**One-block-per-tile approach**: The LLQS array creation process is the same as the one described above but each thread block works on a different tile. Each kernel has one extra loop that iterates through all the bitplane rasters of a tile.
**Kernel 3 - Last Level of the Pyramid Generation:** The third kernel takes as input the LLQS array of each bitplane, and creates the last-level of the pyramid array for each bitplane.

*Multi-block-per-tile:* Each block has 512 threads arranged in one dimension; each thread reads one last-level quadrant value and generates a signature (00 if the quadrant value has only 0s, 11 if the quadrant value has only 1s, and 01 if the value has a mix of 1s and 0s). The generated signatures are stored in a shared temporary array, and after all the threads have executed, these signatures are combined together four by four into chars. This ensures that four consecutive last-level quadrants are stored as siblings.

*One-block-per-tile:* The process to generate the last level is the same as described above, but each block works on a separate tile. As a result, an extra outer loop is added to iterate through all the bitplanes of the tile.

**Kernel 4 - Pyramid Generation:** The fourth kernel takes as input the last-level of each bitplane pyramid array and generates the rest of the pyramid. Each thread in the 512x1 threadblock reads one short and generates a 2-bit signature. This kernel generates the tree level-by-level, with all the threads working on one level at once. At each level we examine the signatures of each four sibling nodes on the level immediately below, and store the parent 2 bit signature in the current level.

*Multi-block-per-tile:* Each block works on one bitplane of a single tile, thus only 16 blocks are launched at one.

*One-block-per-tile:* Each block works on a different tile, thus requiring an extra outer loop to process the tile from 1st bitplane pyramid up to the 16th bitplane pyramid.

The parallelization scheme that was chosen is Process Collectively and Loop (PCL) (described in more detail in [27]). This scheme is named this way because the group of working threads processes a set of elements by looping. For each loop, the threads collaborate to process only a portion of this set. In this scheme we have NumThreads CUDA threads in a 1 dimensional block (where MaxThreadNum is a multiple of 4) with indices i in the range 0 to i<MaxThreadNum> that will be working to build the pyramid array. When constructing each level, all the threads execute a loop with NumElementsInLevelL/NumThreads steps. For example, in Figure 7 we have 4 threads numbered from 0 to 3, each one of them is in charge of reading the entries from the matrix in the bottom of the figure that have their same number. In the first iteration of the loop, the four threads each read its corresponding entry from the first quadrant (shown shaded in the figure). In the second iteration of the loop, each thread reads its corresponding entry, but from the second quadrant in Z-order (the one just below the first one), and so on. Since consecutive threads read consecutive elements in memory, the hardware can coalesce those memory accesses, resulting in faster execution times [6]. Based on the values read, each thread then writes the correct values for level l.

**Kernel 5 - Compression of the BQ-Tree and LLQS array:** This step compresses the pyramid and llqs array. As it has been mentioned previously in subsection 3.2.1, to eliminate the redundancy and compress the pyramid array we will remove sub-quadrants with signatures 0x00 and 0xFF

*Multi-block-per-tile:* The parallelization scheme chosen uses three steps. First, we divide the pyramid and the llqs array uniformly into segments, such that each CUDA thread block

---

**Figure 6:** Major steps for the encoding of raster data using BQ-Trees on GPGPUs. Items on the right represent the work of the kernels and those on the left represent the data generated after each step. The last-level quadrant size is set to 4*4.

**Figure 7:** The Process Collectively and Loop parallelization scheme used to construct the pyramid array.
(launched with 256 threads) will be in charge of processing one of these segments. Each thread is assigned a subset of quadrants of the segment. Then for every one of the quadrants in its subset, if that element has uniform bits, then that thread stores 0 in a Boolean array (of the same size as the pyramid array), and 1 otherwise. Second, a parallel prefix scan is applied over that Boolean array to shift the pyramid and the llqs array positions over those elements that we wish to remove.

One-block-per-tile: We start by assigning each block to a unique tile, then each thread in the thread block read a single quadrant from the llqs or the pyramid array, if the quadrant is uniform, then the value 0 is stored in a Boolean array (of the same size as the number of threads in the block), otherwise value 1 is stored. After all the threads have read a single value, a parallel scan is performed on the Boolean array in order to generate the new positions of non-uniform quadrants.

6. PERFORMANCE STUDIES

In this section, we present our experimental studies comparing the raster image compression performance results obtained from the four implementations: 1) Our GPGPU encoding using BQ-Trees and multiple blocks per raster tile (Multi-block-per-tile); 2) Our GPGPU encoding using BQ-Trees and one block per raster tile (One-block-per-tile); 3) Our Multi-core CPU encoding using BQ-Trees; and 4) Multi-core CPU encoding using the popular zlib compression library [41].

6.1. Datasets and Experiment Setup

The experiments were performed on the NASA MODIS (Moderate Resolution Imaging Spectroradiometer) raster datasets for Africa (referred to as B1 throughout the paper) [35], North America (referred to as B2) [36], and Asia (referred to as B3 [37]. The datasets contain 17352x1700 cells for B1, 22,658x15, 586 cells for B2, and 17352x16700 cells for B3 and have a bit depth of 16 bits (refer to Figure 8). We used these rasters because they are popular and are big size datasets for environmental science applications. They have increasing sizes from B1 to B3 in order to observe the impact of the data size on the CUDA implantation as well as the multi-core CPU implementations. Figure 8 shows the NASA MODIS raster data that we used.

![Figure 8: NASA MODIS raster dataset (raster with depth 16 bits and 17352*16700 cells (dataset B1), 22,658*15,586 cells (dataset B2), 17352*16700 cells (dataset B3)](image)

We can see that the rasters covers mostly oceans (which correspond to the cell value NO_DATA). Also, it is the case that 75.8% of the cells have values less than 4096; so for around three quarters of the total number of cells, they have the four most significant bits equal to 0. These facts lead us to expect that BQ-Trees will achieve a good compression ratio because the bitplane pyramids containing those significant bits will have large areas with uniform bits (all 0s or all 1s), thus allowing to prune larger chunks of the pyramid. The GPGPU hardware used is described in Section 4.2, we also use two Intel Xeon E5520 quadcore chips with Simultaneous Multithreading enabled, and finally, we use version 1.2.8 of zlib to compress the data.

In this study, we first compare the times for the BQ-Tree GPGPU implementations (BQ-Tree GPGPU Multi-block-per-tile and GPGPU Single-Block-per-Tile) with a BQ-Tree CPU implementation running on 16 threads (Multi-Core BQ-Tree), and with a 16 threads implementation of zlib (Multi-Core zlib). The measurements of this comparison were done in terms of wall clock time where we start the clock as soon as the data transfer to the GPGPU starts and stop it as soon as the GPGPU finishes its computations. We do not include the time to bring the data back to the host main memory due to the fact that once the data has been encoded, the intention is to process queries on it directly on the GPGPU. For multi-core CPU implementations, both for BQ-Tree and zlib, we start recording times after the data is loaded in memory, the reason being that we are more interested in the compression times and also because for online processing such as visualization, the data need not be loaded, it is generated on the fly.

In Table 1 we summarize the GPGPU times to encode the NASA MODIS rasters divided into 1024x1024 tiles into a set of BQ-Trees. From Figure 9, we can see that the BQ-Tree implementations outperform the zlib multi-core implementation with 16 threads on average by a 1.1x factor. This shows that the BQ-Tree is a better choice in terms of speed for losslessly compressing raster datasets compared to the popular zlib compression library. Now we compare the different implementations of the BQ-Tree; the GPGPU one-block-per-tile implementation, the performance is improved by a 3x factor compared to the multi-core CPU implementation with 16 threads. The GPGPU multi-blocks-per-tile has results similar to using the multi-core CPU BQ-Tree Implementation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GPGPU Multi-block-per-tile</th>
<th>Multi-block-per-tile</th>
<th>Multi-core CPU</th>
<th>Zlib CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1~465M</td>
<td>1823 ms</td>
<td>576 ms</td>
<td>1506 ms</td>
<td>6286 ms</td>
</tr>
<tr>
<td>B2~597M</td>
<td>2344 ms</td>
<td>740 ms</td>
<td>1848 ms</td>
<td>9367 ms</td>
</tr>
<tr>
<td>B3~597M</td>
<td>2949 ms</td>
<td>895 ms</td>
<td>2233 ms</td>
<td>14854 ms</td>
</tr>
</tbody>
</table>

Table 1: GPGPU vs. multi-threaded CPU vs. zlib
The difference in performance can be explained by analyzing the time used for different kernels on both GPGPU implementation. For a tile size of 1024x1024, we notice that for the one-block-per-tile, the percentage of time spent on the kernels to generate the last level of the pyramid as well as the whole pyramid is significantly lower than the percentage of time spent on the same kernels on the multi-block-per-tile implementation. The reason can be found at the GPGPU grid dimension allocation; the one-block-per-tile implementation initializes a number of blocks equal to the number of tiles in the raster, and in the case of a tile size of 1024x1024 the amount of tiles for all datasets on average is 372 and the GPGPU hardware can launch up to 128 blocks, this means that with this implementation we are fully utilizing the available processing capability. One the other hand, for the multi-blocks-per-tile implementation, during the generation of last level of the pyramid and the rest of the pyramid, all the blocks work on one tile, and each block is assigned to a different bitplane. For a raster of depth 16, only 16 blocks will be launched, which is a very low amount compared to what could be achievable on the GPGPU hardware.

According to the results of our experiments, the GPGPU implementation provide the best compression times for big geospatial data, particularly the one-block-per-tile implementation outperforms the multi-block-per-tile implementation. Because its design allows the full utilization of the GPGPU hardware. As a conclusion, the GPGPU implementation of the BQ-Tree compression technique achieves the best compression times. However, the programmer has to choose a design that will maximize the GPGPU resources by taking into account not only common GPGPU issues (Refer to Section 4.4 and 5.2), but also subdividing the work in a manner that will use as many thread blocks as possible.

7. SUMMARY AND CONCLUSIONS
In this paper we have first presented the main issues that state-of-the-art compression techniques should address: compression ratio, compression speed, spatial locality, as well as fast spatial access. Then we classified popular techniques applied to geospatial raster data based on those issues. Furthermore, we presented a parallel scheme to encode raster geospatial data on GPGPUs using BQ-Trees. The key ideas for the encoding scheme are: i) using a variation of dictionary encoding to eliminate redundant information, and ii) storing neighboring nodes consecutively in memory so that the hardware may coalesce the memory accesses to those nodes.

The experiments performed show that encoding the 16-bit NASA MODIS rasters with sizes in the range between 470MB and 580MB can be done in less than a second on an Nvidia Fermi C2050 GPU card. This represents a speedup of 3X with respect to our dual quadcore CPU (each an Intel Xeon E5520) implementation running with 16 threads, and a 12X speedup with respect to a zlib implementation. For future work, given that a BQ-Tree can serve both as a compression mechanism and as an index, we would like to study efficient GPGPU algorithms to process geospatial queries using BQ-Trees as index structures. As a second topic, even though we are comparing our implementation against our GPGPU implementation of BQ-Trees, it would be particularly interesting to develop a hybrid CPU-GPGPU implementation that makes use of the advantages offered by the two architectures. For example, we can use a CPU implementation to generate the upper levels of the pyramid array which offers less parallelism since there are fewer entries in the upper levels), and at the same time, use a GPGPU for the deeper levels that offers far more opportunities for parallelism. We will also explore the scalability of this implementation by using a multi-GPU setting or large GPU clusters.

8. ACKNOWLEDGMENTS
This work is supported in part by the National Science Foundation under Grant No.1302439 and 1302423.

9. REFERENCES
7. Bakkum, P. and Skadron, K., Accelerating SQL database operations on a GPU with CUDA. in GPU’10. Proc. of the 3rd workshop on GPGPUs, 94-103