# Parallel Compression and Indexing of Large-Scale Geospatial Raster Data with GPGPUs

Nathalie Kaligirwa, Eleazar Leal, Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK USA
{Nathalie-kaligirwa-1, eleal, ggruenwald}@ou.edu

Jianting Zhang, Simin You
School of Computer Science
City College of New York
New York, NY USA
{jzhang@cs.ccny, syou@gc}.cuny.edu

*Abstract*— **Global remote sensing and large-scale environment modeling have generated vast amounts of raster geospatial data. Performing spatial queries over such data has applications in many domains, such as climate impact studies, water and wildlife management, and urban planning. Processing those queries is greatly facilitated by the existence of spatial indices. However, I/O transfer is still a major bottleneck in the overall system performance. One of the solutions to this issue is to compress data before sending it over the I/O channel. Therefore, a lossless compression technique that also supports spatial indexing to improve query response time is highly desirable. To fill this gap, in this paper we propose two parallel GPGPU algorithms, called Multi-Block per Tile (MBPT) and One-Block per Tile (OBPT), to compress and index large-scale geospatial raster data using BQ-Trees. Experiments comparing our best performing proposed algorithm, OBPT, against HFPaC, a state-of-the-art geospatial parallel GPGPU compression algorithm, using three real datasets of satellite images, show that our algorithm achieves a compression time speedup of up to 2X, and a 2.5X increment in compression ratio. OBPT also yields a comparable average spatial query response time to HFPaC.**

**Keywords-raster data compression; raster data indexing; query processing on raster data; query processing on GPGPUs**

## I. INTRODUCTION

Nowadays, as a result of the advances in global remote sensing technologies, there is an ever-increasing amount of raster spatial data being created. For example, the next generation of weather geostationary satellites called GOES-R [19] will generate 288 global coverages per day at a temporal resolution of 5 minutes at each of its 16 bands. The amount of data collected by them will be very large: these satellites use a spatial resolution of 2 Km., so that each coverage and band combination generates a quarter of a billion cells. Raster data like these have applications in GIS applications to perform analyses such as measuring temperature changes in time over a certain geographic area [3], monitoring deforestation [13], or land use over an area [9]. These analyses transform raster data into knowledge by performing computationally-intensive spatial queries and simulations.

There are multiple issues concerning the management of raster spatial data. One of these issues has to do with the fact that computer processor speed has increased steadily over the years, while the I/O subsystem has experienced very small performance improvements over the same period. This means that when performing spatial queries and simulations on raster data, the main bottleneck in performance comes from the I/O transfers. This issue is made worse by the large volumes of raster data. Therefore, there is a need for reducing the amount of I/O transfer in order to scalably process these queries and simulations. One way of accomplishing this is through data compression: by reducing the size of these raster data, one can effectively trade space and I/O transfer time for the time spent encoding/decoding the data. An implication of this is that compression and decompression algorithms need to be efficient and able to deal with Big Data [5]; therefore, to scalably cope with the volume of these data, there is a need for parallel computing.

Another issue related to the management of large-scale raster spatial data is that data should be represented in a way that allows the design of scalable spatial query processing algorithms. If a proposed large-scale raster data management technique uses compression as a means to deal with the I/O transfer issue, then this technique should avoid decompressing all the data every time a spatial query is issued.

Existing classical serial techniques like [2], [4], [10], and [23], or modern GPU implementations like [1], [14], and [15] can be used to compress the raster data; however, they do not address the issue of allowing the design of scalable spatial query processing algorithms because they require the whole raster to be decompressed in order to process spatial queries.

In this paper, we propose two parallel GPGPU algorithms, called MBPT and OBPT, for compressing and indexing large-scale raster geospatial data using the BQ-Tree [22] [11]. These two techniques address all issues of large-scale raster geospatial data management: they are designed to cope with Big Data thanks to the fact that they are designed to run on massively parallel processors (GPGPUs); they address the issue that relates to the slow I/O subsystem by compressing the raster data, and they address the issue of providing efficient support for spatial queries. We then conduct an experimental study comparing the two proposed algorithms with a GPGPU state-of-the-art technique, called HFPaC [6], for height-field data compression using three real-world datasets. More precisely, the contributions of this paper are the following:

- We propose two GPGPU algorithms called Multi-Block per Tile (MBPT) and One-Block per Tile (OBPT) for raster compression and indexing using the BQ-Tree data structure. These algorithms address the issues outlined before

that concern the management of large-scale raster geospatial data.

- We present an experimental comparison between these two algorithms and a state-of-the-art parallel GPGPU technique specifically designed for compressing geospatial height-field raster data called HFPaC [6]. This experimental comparison uses compression ratio, compression time and also average spatial query processing time as performance metrics.

The rest of this paper is organized as follows. Section II presents related work; Section III presents a review of the BQ-tree; Section IV contains the description of the two proposed parallel GPGPU BQ-Tree encoding techniques Multi-Block per Tile (MBPT) and One-Block per Tile (OBPT); Section V presents the experimental evaluation of the two proposed techniques, and the comparison against HFPaC; finally, Section VI provides conclusions.

## II. Related Work

There exist several serial algorithms for encoding all kinds of bit sequences and that can be specifically applied for compressing raster data. Among such techniques there are compression techniques like Huffman Coding [10], Arithmetic Coding [21], Run-length encoding [17], Lempel-Ziv Encoding [23], Burrows-Wheeler Transform [4], Word-aligned Hybrid (WAH) [2]. All these techniques have in common that they address the issue of compressing spatial data and thus allow dealing with raster data scalably. Their most notable disadvantage is that in order to issue spatial queries, the encoded data (i.e. the output of these algorithms) must first be decompressed, and then, after the query is processed, the data may need to be compressed again. There exist other serial techniques for dealing with raster data like the R-tree (and its derivatives) [8], and the Quadtree [16] that address the issue of spatial query support, but do not compress the data. Another technique for dealing with raster data is the ISOBAR pre-conditioner [18], which is a lossless compression technique that divides the data into byte chunks and then identifies which of these chunks should be compressed to maximize the compression ratio. The advantage of ISOBAR is that since the chunks are independent, it should be possible to design a spatial query processing algorithm that does not need to decompress all the data in order to run.

HFPaC [6] is a parallel GPGPU algorithm that offers both lossless and lossy compression for height-field raster data. Height-field data can be considered as a real-valued function defined over a two-dimensional grid such that the value at each grid cell corresponds to the height of the terrain at that grid cell. HFPaC estimates the content of a grid cell by using Bézier surfaces. A Bézier surface is a type of mathematical spline, i.e. it is a smooth mathematical function that can be used to approximate a surface. To compress height field data, HFPaC fits a Bézier surface to those data by finding the appropriate control points. Since the control points completely define a Bézier surface, then by only keeping the control points, HFPaC can approximate the original data. Additionally, since the control points are far fewer than the original data points, by only keeping the control points HFPaC can provide lossy compression. To provide lossless compression, HFPaC stores the error between the value estimated by the Bézier surface and the original value in the height field data. However, for this technique to achieve a high compression ratio the data must be smooth, as is the case of height field data. Otherwise, the Bézier approximations can significantly deviate from the true values and lead to larger errors.

LZSS [14] is a derivative of the well-known compression algorithm LZ77 [23]. The disadvantage of this technique is that, despite addressing the issue of data compression, it does not allow spatial queries on two-dimensional data because it only works in a single dimension.

GPU-WAH [1] is a GPGPU compression algorithm that extends WAH [2]. GPU-WAH compression is executed in two major steps: bitmap extension and compression. However, WAH does not address the spatial aspect of data because WAH compresses data by rearranging each word in the input bitmap, and this requires a full decompression of the data before running spatial queries.

Another technique for managing large-scale raster data is the parallel GPGPU R-tree [20]. The approach taken is to use a linear array instead of pointers to represent the R-Tree, which enables memory coalescing and efficient data transfer from the host to the device. In addition, each non-leaf node of the parallel R-Tree is represented as a tuple containing the minimum bounding box, a position *pos* indicating the position of the first child, and a length *len* with the number of children. The disadvantage of this technique is that, just like with the classical R-tree, it does not address the issue of compressing the raster data.

## III. The BQ-Tree

In this section we present a review of the BQ-Tree [22] [11], which is a data structure for large-scale raster data compression amenable for parallel GPGPU architectures. Our proposed algorithms, MBPT and OBPT, introduced later in Section IV, are GPGPU techniques designed for encoding large-scale raster geospatial data using this data structure.

### *1) Overview*

Suppose that we are given a bitmap $B$ of size $2^m \times 2^m$, with $m > 0$, a bit depth value $D > 0$ (the number of bits in each entry of the bitmap), and a last-level quadrant of size $2^q \times 2^q$ such that $q \leq m$. The bitmap $B$ can be encoded using $D$ BQ-trees, with one tree corresponding to each bitplane of the bitmap. Each one of these BQ-Trees contains a Pyramid Array and a Last-Level Quadrant Signatures array (LLQS).

The BQ-Tree's pyramid array is built out of a pyramid having exactly $m-q+1$ levels, where the top level (containing only the root) is level number 0 and the bottom level is level number $m-q+1$. Every entry in level $i \neq m - q$ contains

exactly four children entries in level $i+1$. Therefore, for any $0 \leq i \leq m-q$ the $i^{th}$ level in the pyramid contains $2^i \times 2^i$ elements.

Each one of the $D$ bitplanes can be subdivided into sub-bitplanes, called "last-level quadrants," of size $2^q \times 2^q$ (see Figure 1, which shows last-level quadrants for two different values of $q$: the one in the left uses $q = 1$ and the one in the right uses $q = 2$). There is a one-to-one correspondence between the last-level quadrants and the nodes in the last-level of the pyramid; this correspondence is shown in Figure 2. In this figure, we see that in the matrix corresponding to level 2, the entry in the first row second column is 01 because its corresponding last-level quadrant (highlighted with a circle in the last-level matrix) contains both 0s and 1s.

A node in the last-level of the BQ-Tree has value 00 if all the elements of its corresponding last-level quadrant are 0s, 11 if all the elements of its corresponding last-level quadrant are 1s, and 01 if its corresponding last-level quadrant contains both 0s and 1s. See Figure 1, where the matrix corresponding to level 2 has in its (0,0) entry (first row first column) the value 11 because its corresponding last-level quadrant entries (at positions (0,0), (0,1), (1,0), (1,1) in level 3) all contain 1s.

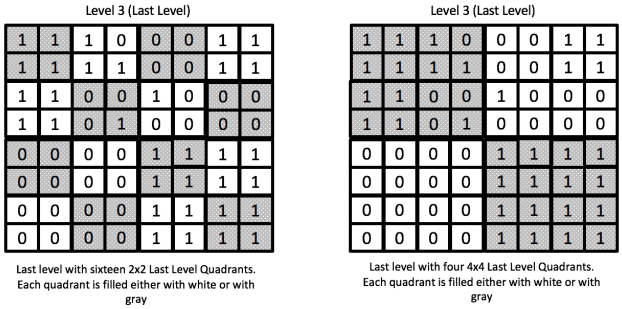


Figure 1. Choice of size for the Last-Level Quadrants (LLQS)

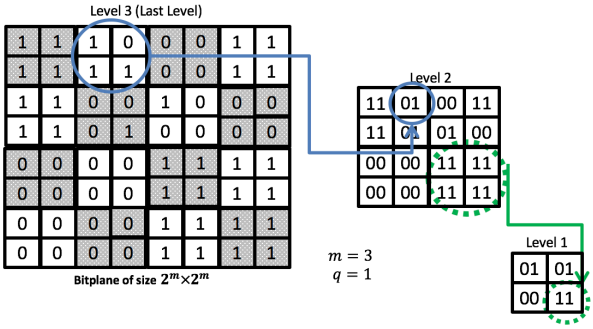


Figure 2. Levels of a BQ-Tree

A node $n$ at level $0 \leq i \leq m-q-1$ contains the value 00 or 11 if its four children all have values 00 or 11, respectively. If $n$ has at least two children with different values, then the value for $n$ is 01. For example, in Figure 2 we see that entry (1,1) of the level 1 matrix is 11 (marked with a striped circle) because its four children in level 2 are all 11.

### 2) BQ-Tree Construction

We now explain how to construct a BQ-Tree for encoding a given a bitplane. The BQ-Tree is based on two key ideas: the first one is the linearization of all the nodes in the bitplane. This means that if four nodes are all children of the same parent, then they are all placed in adjacent memory locations. The second key idea is the use of a variation of dictionary encoding [21]: 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).

The BQ-Tree encoding of the bitplane shown in Figure 1 is illustrated in Figure 3, which shows the pyramid array, and Figure 4, which shows the LLQS array. We now explain how to construct each of these arrays.

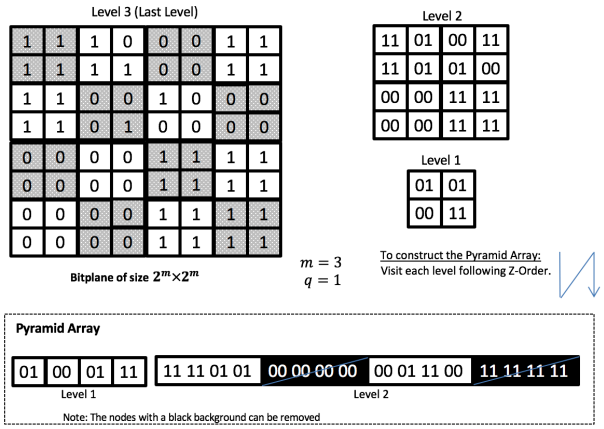


Figure 3. Construction of the Pyramid Array

### a) Constructing the Pyramid Array

To construct the pyramid array, we proceed to enumerate all the entries of level 1, then all entries in level 2, and so on until we reach level $m$-$q$. As we can see in Figure 3, we visit all entries in Level 1 following a Z-order [12], and write them to the pyramid array in the same visiting order. For example, the entries in Level 1 listed in Z-order are 01 00 01 11, and are the first entries in the pyramid array. Then for Level 2, its entries listed in Z-order are 11, 11, 01, 01, 00, 00, 00, 00, 00, 01, etc. which follow the entries of Level 1 in the Pyramid Array.

In the above enumeration some redundant nodes can be pruned off of the BQ-Tree. We illustrate this with an example. In Figure 3 we see that the four entries of Level 2 located at positions (2,2), (3,2), (2,3) and (3,3) (all containing the value 11) all share the same parent node which is the entry located at position (1,1) of level 1's matrix. Since the parent node of that group of four entries has value 11, then there is no need to output any of the codes corresponding to positions (2,2), (3,2), (2,3) and (3,3) because the parent entry already tells us the values of all its children. Figure 3 contains another example illustrating the pruning of redundant information from the Pyramid Array, where we see a sequence of bits with a black background that has been crossed out. This is because in level 1 there is an entry 00 (the bottom left) whose four children are also 00.

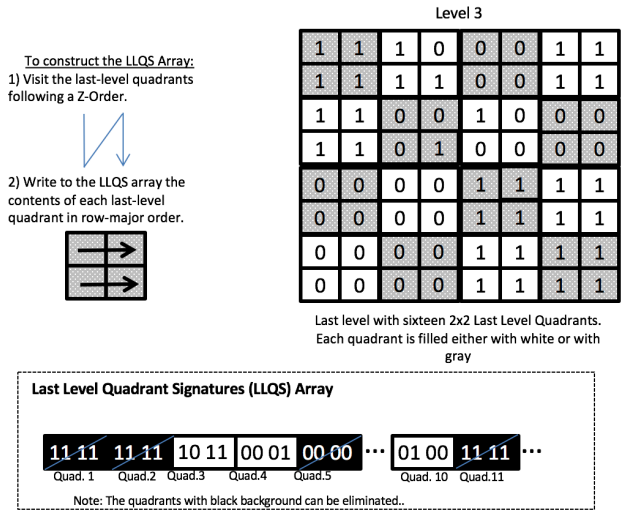


Figure 4. Construction of the LLQS Array

*b) Constructing the LLQS Array*

To construct the LLQS array, we visit the last-level quadrants of the bitplane following a Z-order. Each time we visit a last-level quadrant, we output its contents to the LLQS array in row-major order. For example, in Figure 4 we see to the right the set of last-level quadrants, and in the bottom we see the LLQS array. The first last-level quadrant we visit is the one shaded in gray at the top-left corner containing 1s. We then write four 1s to the LLQS array. Then, the next last-level quadrant we visit is the one below the one at the top-left corner that is not shaded. We then output its four 1s to the LLQS array. The third last-level quadrant is the one to the right of the top-left quadrant. We output its entries to the LLQS array in row major order as 10 11, which is what we see in Figure 4. We continue in this fashion until we visit all last-level quadrants.

Once all last-level quadrants have been output to the LLQS array, we can prune some information off of the LLQS array since we know that the last-level of the pyramid will tell us if a last-level quadrant is uniform (all 1s or all 0s) or not. This is why in Figure 4 we prune the bits corresponding to quadrants 1, 2, 5, etc.

## IV. Proposed BQ-Tree Encoding GPGPU Algorithms

In this section we present our two parallel GPGPU techniques, OBPT and MBPT, for compressing and indexing large-scale raster geospatial data. This section concludes describing a GPGPU algorithm to perform spatial range queries over the compressed rasters obtained as output.

*1) One block per tile Algorithm (OBPT)*

A raster is split into tiles of uniform size (Line 1 in the OBPT procedure) and then all these tiles are transferred from the host's main memory to the GPGPU's global memory (Line 2). The GPGPU then assigns separate thread blocks to process each of the tiles in of the raster data (Line 4).

After this is done, the compression process starts with decomposing the tiles (each a 16 bit bitmap) into *16* distinct bitplane bitmaps (Line 5). Then, the GPGPU builds for each of the *16* bitplanes the corresponding LLQS array (Lines 6 and 7). Next, each LLQS array is used to generate the last level of the pyramid and then the whole pyramid (Lines 8 and 9). At the end of this algorithm, the output is *16* BQ-Trees.

```
procedure OBPT(16 − Bit Raster r, tile size ts)

1   tiles ← split_into_tiles(r, ts) // Divide the raster into tiles
2   gpu_tiles ← Copy all tiles into the GPU's global memory
3   BQ-Trees ← ∅
4   For each tile in gpu_tiles do in parallel at each thread block
5       Bitplanes ← Decompose tile into 16 bitplanes
6       For each bitplane in bitplanes
7           LLQS ← Build LLQS array for bitplane
8           PA ← Fill the last-level of the pyramid array
9           PA ← Fill remaining levels of the pyramid array(PA)
10          BQ-Tree ← (LLQS,PA)
11          BQ-Trees.add(BQ-Tree)
12  End for
13  Return BQ-Trees
```

*2) Multiple blocks per tile Algorithm (MBPT)*

In this section, we present a description of the MBPT. MBPT is a parallel algorithm which is designed for compressing geospatial raster data by transferring only one tile at a time and using all GPGPU resources on one tile. This technique can increase the compression time for geospatial raster data with few but very large tiles. The process follows three major steps:

First, before the compression steps are undertaken, a tile is transferred from host memory to GPGPU memory. After a tile *i* is compressed, the next tile *i+1* is loaded from host memory to the GPGPU's global memory and the compression process is repeated. This mechanism is performed for each tile until the whole raster image is processed.

```
procedure MBPT(16 − Bit Raster r, tile size ts)

1   tiles ← split_into_tiles(r, ts) // Divide the raster into tiles
2   BQ-Trees ← ∅
3   For each tile in tiles  // Tiles processed sequentially
4       gpu_tile ← Copy tile into the GPU's global memory
5       // Use all threads and thread blocks to process each tile
6       Bitplanes ← Decompose tile into 16 bitplanes
7       For each bitplane in bitplanes
8           LLQS ← Build LLQS array for bitplane
9           PA ← Fill the last-level of the pyramid array
10          PA ← Fill remaining levels of the pyramid array(PA)
11          BQ-Tree ← (LLQS,PA)
12          BQ-Trees.add(BQ-Tree)
13  End for
14  Return BQ-Trees
```

Then, the compression process starts with decomposing the tile (which is a 16-bit bitmap) into 16 distinct bitplanes. After this, for each of the bitplane bitmaps the LLQS array is constructed; at the end of this step, there are 16 LLQS arrays. Then, each LLQS array is used to generate the last level of

the pyramid and then the whole pyramid. The last two steps compress the pyramid and the LLQS array for each bitplane bitmap. At the end of this algorithm, the output is 16 BQ-Trees. Finally, the 16 BQ-Trees are transferred back to the host's memory.

*3) Spatial Range Query Algorithm*

In this subsection we present definitions for a query window and for spatial range queries. Then, we describe a GPGPU algorithm for processing spatial range queries over the output raster from OBPT and MBPT.

**Definition (Query Window)**: Given an $m \times n$ real matrix $R = [r_{ij}]$, the window $W(m_1, m_2, n_1, n_2)$ is the set of all integer pairs (*i,j*) such that $1 \leq m_1 \leq m_2 \leq m$ and $1 \leq n_1 \leq n_2 \leq n$.

**Definition (Spatial Range Query)**: Given an $m \times n$ real matrix $R = [r_{ij}]$ and a window $W(m_1, m_2, n_1, n_2)$, the spatial range query *SRQ (R,* $W(m_1, m_2, n_1, n_2)$) is defined as the set $\{((i,j), r_{ij}) \mid (i,j) \in W(m_1, m_2, n_1, n_2)\}$ where $r_{ij}$ is the *ij* entry in matrix *R*.

The spatial range query is implemented following the filter-and-refine paradigm as follows:

**Filtering step**: Given a spatial range query with a window *W*, we first determine which tiles are covered by the window *W*. The filtering step returns all raster cells from the candidate tiles which have their Z-order within the Z-order of the boundaries of window *W*. This step is performed in parallel on the GPGPU.

**Refining step**: The values returned from the filtering steps are passed into the filtering steps, where the coordinates *i, j* of each raster cell are compared with the coordinates of the window *W*. This step provides the final results and is performed on the CPU.

## V. Performance Analysis

### A. Hardware and Software Setup

For our experiments we used an SGI Octane III workstation equipped with four Nvidia Fermi C2050 GPU cards, two quadcore chips, and 48GB of RAM.

### B. Datasets

The experiments were performed on three 16-bit NASA MODIS (Moderate Resolution Imaging Spectroradiometer) raster datasets corresponding to Africa $17{,}352 \times 16{,}700$ cells), North America ( $22{,}658 \times 15{,}586$ cells), and Asia ($17{,}352 \times 16{,}700$ cells) [7]. These rasters have a relatively large size and are commonly used in environmental science applications.

### C. Competing Technique

We compare our proposed techniques with HFPaC [6]. HFPaC is a parallel GPGPU compression algorithm designed for compressing height field spatial data using Bézier curves and surfaces. HFPaC works by first dividing the raster data into tiles of size $(2^n + 1) \times (2^n + 1)$, with $7 \leq n \leq 12$, and each tile is processed independently. Secondly, each tile is divided into segments of size $(2^n + 1) \times (2^n + 1)$ with $2 \leq n \leq 5$. For each segment, Bézier control points are calculated and stored in the first layer of the compressed format. Then, the row and column indices of each height field cell are used as the *x* and *y* axis values for the Bézier function (which uses the Bézier control points calculated) to generate an approximate value for the cell. The second and third layers contain the error incurred by the Bézier approximation with respect to the true value of the cell. The number of bits to represent the error is variable, but in order to achieve a lossless compression, 8 bits are used. So for this algorithm, we analyze its performance using tiles and segments of varying sizes. Table 1 presents a feature comparison between HFPaC and the BQ-Tree.

Table 1. Feature Comparison of Techniques

| Feature | *HFPaC* | *BQ-Tree* |
|---|---|---|
| **Lossy or Lossless** | Can be either one depending on the parameters | Lossless |
| **Type of Data** | Height Field data | Raster data |
| **Byte Partitioning** | *b* bits (after compression) | Bit-level (before compression) |
| **Data partitioning** | Tiles of size $(2^n + 1) \times (2^n + 1)$ where $7 \leq n \leq 12$ | 1024×1024 or 4096× 4,096 tiles |

### D. Parameters

In the experiments performed, there are 4 dynamic parameters whose values are changed in order to study their impacts on the evaluation metrics. These dynamic parameters are the raster dataset (whose default value is the North American Raster because it is the one with an intermediate size, when compared to the other two rasters), the tile size, the HFPaC segment size (which is a user-defined parameter for HFPaC only, so it does not impact OBPT nor MBPT), and the number of spatial queries issued against the output of both techniques.

### E. Evaluation Metrics

To measure the performance of our proposed BQ-Tree encoding algorithms, we employ the metrics described below.

**Compression Time**: This metric is the total time elapsed to encode a whole raster. When comparing our proposed encoding algorithms (OBPT and MBPT) with each other, this metric includes the time employed to transfer the input raster data from the host to the GPGPU device, and then to transfer the output, i.e. the compressed raster, back from the GPGPU to the host. Nonetheless, when comparing our proposed algorithms against HFPaC, the compression time only includes the time consumed during the input raster transfer

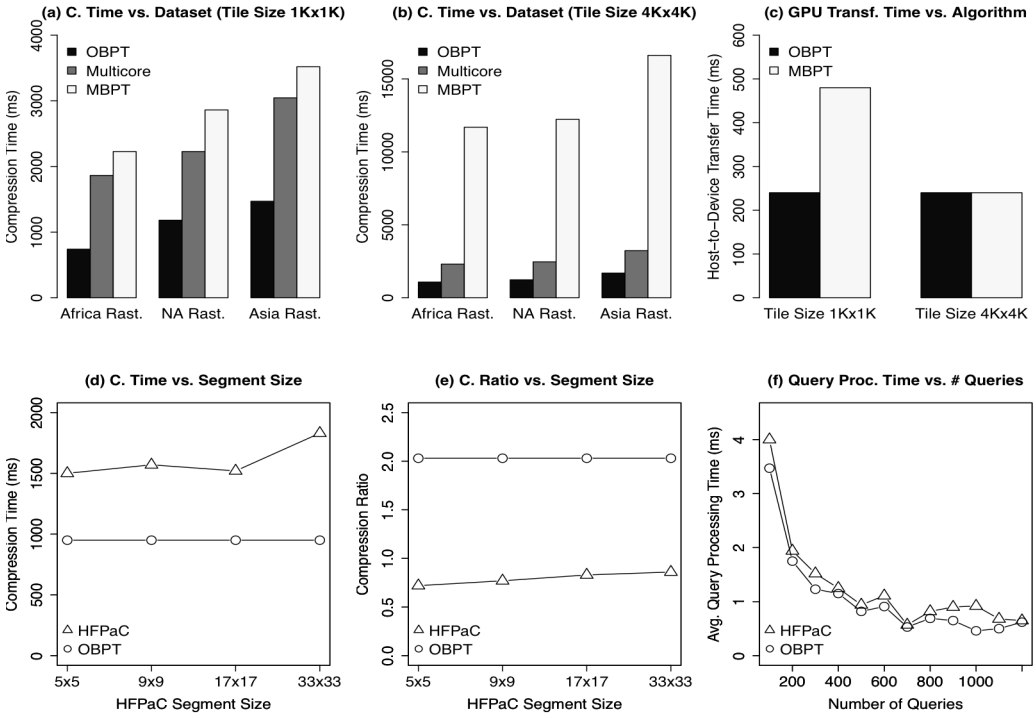


Figure 5. Experimental Results Comparing OBPT vs. MBPT and OBPT vs. HFPaC

from the host to the GPGPU, and the GPGPU kernel time. The compression time does not account for the time employed to transfer the output from the GPGPU back to the host because the HFPaC implementation does not allow a fair measurement of the time for the data transfer from the GPGPU to the host.

**Compression Ratio**: The compression ratio is calculated by dividing the original input raster file size by the size of the output. More formally, compression ratio: (size of input raster)/(BQ-Tree + metadata). In the case of HFPaC, the compression ratio is computed as: (size of the input raster) /(size of layer 1 + size of layer 2 + size of layer 3).

**Average Query Processing Time**: The average query processing times were obtained by running several random spatial range queries, which were described in Section IV.3. The total query processing time includes both the decompression and the query processing time. The decompression time does not include the time to load the compressed data from the host to the GPGPU because the competing algorithm is implemented in such a way that does not allow a fair comparison of the transfer time from the host to the GPGPU.

### *F. Experimental Results*

In this subsection we evaluate the impacts of our experiment parameters on the chosen performance metrics. Initially we compare our two proposed GPGPU raster compression algorithms, OBPT and MBPT, against our multicore implementation. After performing these experiments, we determine the best performing algorithm out of these in order to compare it against HFPaC.

#### *1) Impact of the Raster File Size in Compression Time*

In a sequential algorithm, the compression time is expected to increase as the dataset increases. However, for GPGPUs, execution time will increase only after the utilization of all resources reaches 100%. Because the Africa raster is the one with the smallest file size, and the Asian raster is the largest one, then the impact of the raster file size on the performance of the techniques can be determined by running all the algorithms with the different rasters.

In Figure 5a we observe that for a tile size of $1024\times1024$ the OBPT algorithm has the shortest compression times (around twice as fast as MBPT and multicore) for all three rasters when compared with MBPT and our multicore implementation of BQ-Tree encoding. The reason for this behavior is that the OBPT algorithm achieves an optimal resource allocation. Indeed, the number of blocks allocated in the kernel launch (289, 368, 459 for the African, North American and Asian rasters, respectively) far exceeds the maximum number of blocks that can be simultaneously active (which is 8 blocks for a Fermi architecture), so they keep the GPGPU's block queue saturated. Also, the OBPT algorithm assigns more work to each thread than MBPT. In Figure 5b it can be observed that the OBPT algorithm exhibits the same behavior described before with a tile size of $4096\times4096$, which is again due to the fact that OBPT achieves a better resource utilization because it allocates 25, 26 and 35 blocks for the African, North American and Asian

rasters, respectively. The MBPT compression time performance is linear across different dataset sizes, but is much better with a tile size $4096\times4096$. This is because larger tile sizes expose more parallelism that can be exploited by the GPGPU.

*2) Impact of Tile Size on Host to Device Transfer Time*

The impact of the tile size on the host to device transfer can be observed in Figure 5c. In this figure we see that with a tile size of $1024\times1024$ MBPT takes twice as long to transfer the data from the host to the device than it does with a tile size of of $4096\times4096$. The reason for this is that MBPT transfers only one tile at a time from the host to the GPGPU. For example, for the $1024\times1024$ tile size on a raster image of size 771 MB, there will be 368 tile transfers between the host and the GPGPU. On the other hand, the OBPT technique transfers the whole raster image at once to the GPGPU. In case the images are too large to fit on the GPGPU's memory, the data transfer and data compression are executed in multiple parts.

For a tile size of $4096\times4096$, the MBPT takes only 24 loops to transfer the raster image, so that is the reason why it takes fewer time to transfer the data. In general, with MBPT the time spent in memory transfer improves with larger the tiles because bigger tiles imply fewer numbers of tiles, which reduces the number of transfers from the CPU to the GPGPU. On the other hand, for OBPT, the memory transfer is the same regardless of the tile size because the whole raster is sent to the GPGPU all at once.

*3) Impact of Segment Size on Compression Time*

HFPaC requires three input parameters: the size of the tiles, the size of the segment, and the level of error in the compressed format. Since the BQ-Tree is a lossless compression technique, we compare only the lossless version of HFPaC; therefore, we do not need to set the level of error parameter. HFPaC can handle data partitioned in tiles of size $(2^n+1)\times(2^n+1)$, where $7 \le n \le 12$. We study the impact of segment size using two tile sizes $(2^{10}+1)\times(2^{10}+1) = 1025\times1025$, and $(2^{12}+1)\times(2^{12}+1) = 4097\times4097$ because those are closest to the tile sizes supported by the BQ-Tree.

In Figure 5d, it can be observed that the compression time of HFPaC increases as the segment size increases. HFPaC achieves its best performance for a tile size of $4097\times4097$ using a segment size of 5x5. From this figure we also see that OBPT outperforms HFPaC across all segment sizes by a factor of 6X on average.

*4) Impact of the Segment Size on Compression Ratio*

As it has been observed in experiments 1, 2 and 3, OBPT outperforms both MBPT and our multicore implementation of the BQ-Tree encoding algorithm in terms of compression time and GPU transfer time for all three raster datasets and both tile sizes. For this reason, we only compare OBPT against HFPaC in the following experiments.

For OBPT, the compression ratio depends only on the data distribution. For example, if the data contains large regions with uniform values, we are likely to get a high compression ratio. In the case of HFPaC, the compression ratio depends on the size of the segments and the type of data.

In Figure 5e, we see the impact of the segment size in the compression ratio when compressing the MODIS North American raster dataset. We observe that the compression ratio of the OBPT is consistently better than HFPaC's by a factor of 2 on average. In addition to this, we notice that HFPaC's compression ratio for this dataset is smaller than 1, meaning that the compressed output is actually larger than the input raster data, while OBPT's compression ratio for this dataset is around 0.5.

*5) Impact of the Number of Queries on Average Query Execution Time*

The average query compression time was measured by running random spatial range queries and dividing the total query processing times by the number of queries. The experiments are run in many iterations; for the *i-th* iteration, the raster is decompressed once and then *100i* queries are ran one after the other. The total query processing time includes the average decompression time per iteration as well as the average query processing time. The queries were run on the MODIS dataset North American Raster using a tile size of $1024\times1024$ and $1025\times1025$ for OBPT and HFPaC, respectively.

In Figure 5f it can be observed that as the number of spatial queries issued against the output of both techniques increases, the average query processing time diminishes. The reason for this is that for each batch of queries, the data needs to be decompressed only once. Therefore, with larger query batches, the decompression time remains constant, but the time to run all the queries increases and the impact of decompression is reduced as a result. From this figure it can also be observed that the average query processing time of the OBPT is faster than that of HFPaC by a factor of 1.23X on average.

## VI. Conclusions

In this paper we proposed two GPGPU algorithms OBPT and MBPT for large-scale raster geospatial compression, indexing and querying. Our experiments showed that OBPT outperforms MBPT in all datasets in terms of compression ratio and GPGPU transfer time between the host and the device. This is because OBPT has been designed to maximize the GPGPU resource utilization by keeping all blocks and thread blocks busy at all times. We also compared our best-performing GPGPU algorithm, OBPT, against HFPaC, a state-of-the-art height-field compression technique, and showed that OBPT consistently outperforms HFPaC in terms of compression time and compression ratio for the selected raster datasets, while providing a competitive average spatial query execution times.

## ACKNOWLEDGMENT

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

## REFERENCES

[1] Andrzejewski, W., Wremberl, R. GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid. Database and Expert Systems Applications. Lecture Notes in Computer Science 6262, pp. 315-329, 2010.

[2] Anh N. V., Moffat A. Inverted Index Compression Using Word-Aligned Binary Codes. Information Retrieval, pp. 1510-166, 2005.

[3] Bhaskaran, V., Konstantinides, K. Image and Video Compression Standards. Springer Science, New York, 1997.

[4] Burrows, M., Wheeler, D. J. A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.

[5] Cuzzocrea, A. Big data compression paradigms for supporting efficient and scalable data-intensive IoT frameworks. In Proceedings of the International Conference on Emerging Databases, 2016.

[6] Durdevic, D. M., Tartalja, I. I. HFPac: GPU friendly height field parallel compression. Geoinformatica. 17(1), pp. 207-234, 2013.

[7] Global Land Cover Facility (GLCF) MODIS 500m North America Dataset. ftp://ftp.glcf.umiacs.umd.edu/modis/500m/North_America/

[8] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84.

[9] Huffman Coding: An Example. http://www.binaryessence.com/dct/en000080.htm

[10] Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes. In Proceedings of the I.R.E, 1952.

[11] Kaligirwa, N., Leal, E., Gruenwald, L., Zhang, J., You, S. Parallel QuadTree encoding of large-scale raster geospatial data on multicore CPUs and GPGPUs. In Proceddings of BigSpatial 2014.

[12] Mamoulis N. Spatial Data Management. Morgan & Claypool, San Rafael, 2012.

[13] Marvel, L., Hartwig, G. W. A survey of Image Compression Techniques and Their Performance in Noisy Environments. Army Research Laboratory, 1997.

[14] Ozsoy, A.; Swany, M. CULZSS: LZSS Lossless Data Compression on CUDA. In Proceedings of the IEEE International Conference on Cluster Computing, 2011.

[15] Ozsoy, A., Swany, M., Chauhan, A. Optimizing LZSS compression on GPGPUs. Future Generation Computer Systems, vol. 30, pp. 170-178, 2014.

[16] Samet, H. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco, 2005.

[17] Sayood, K. Lossless Image Compression. Introduction to Data Compression. Fourth Edition. Morgan Kaufmann, 2012.

[18] Schendel, R. E., Jin, Y., Shah, N., Chen, J., Chang, C.S., Ku, S.-h., Ethier, S., Klasky, S., Latham, R., Ross, R., and Smatova, B.F, ISOBAR Preconditioner for effective and high-throughput lossless data compression. In Proceedings of the International Conference on Data Engineering, 2012.

[19] Schmit, T.J., Li, J., et al. High-spectral- and high- temporal resolution infrared measurements from geostationary orbit. Journal of Atmospheric and Oceanic Technology, 26(11), pp. 2273-2292, 2009.

[20] Simin, Y, Zhang, J. and Gruenwald, L. Parallel Spatial Query Processing on GPUs using R-Trees. In Proceedings of BigSpatial, 2013.

[21] Solomon, D. Data Compression, The Complete Reference. Springer-Verlag, London, UK. 2007.

[22] Zhang, J., You, S. A Quadtree-Based Lightweight Data Compression Approach for Processing Large-Scale Geospatial Rasters. GIS'11, pp. 457-460, 2011.

[23] Ziv, J., Lempel, A. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23 (3), pp. 337–343, 1977.