UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PARALLEL COMPRESSION AND INDEXING ON LARGE-SCALE GEOSPATIAL RASTER DATA ON GPGPUS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCEMASTER OF SCIENCE

By

NATHALIE KALIGIRWA
Norman, Oklahoma
2015

PARALLEL COMPRESSION AND INDEXING ON LARGE-SCALE GEOSPATIAL RASTER DATA ON GPGPUS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCESCHOOL OF COMPUTER SCIENCE

BY

_____
Dr.Dr. Le Gruenwald, Chair

_____
Dr.Dr. Qi Cheng

_____
Dr.Dr. Changwook Kim

To mom, dad, Alice, Julien and Ronald.

Acknowledgements

I have been working for two years on this research and it has been the most satisfying and challenging academic task I have ever undertaken. As this research is coming to a completion, I have many people to thank for their guidance, support and encouragement.

I would like to first thank my advisor, Dr. Le Gruenwald for taking a chance on me and introducing me to the world of academic research. I appreciate her unwavering support and guidance throughout this journey. I also want to thank the brilliant members of my research group, Dr. Jianting Zhang, Mr. Simin You, and Mr. Eleazar Leal Gonzalez; without their expertise and help I would not have been able to conduct this research. Also, a special mention goes to the National Science Foundation for supporting this research. I would like to thank Dr. Qi Cheng and Dr. Changwook Kim for accepting to review my work and to be part of my committee. A special word of gratitude goes to Mrs. Virginie Perez Woods for always going the extra mile to help me with any administrative, academic and even personal problem.

Finally, I would like to extend special thanks to my family and friends for their constant encouragements. I am grateful for my family's understanding and support over these last two years. I also thank my friends all over the world for keeping my spirits and motivation up.

Table of Contents

List of Tables

List of Figures

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. Additionally, though there have been major advances in computational power, I/O transfer is still 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 data compression technique that not only reduces storage space but also supports indexing to improve query response time is highly desirable. These two issues, compression and indexing, can be efficiently addressed by the BQ-Tree which is an innovative spatial data structure that has been shown to achieve competitive compression time as well as compression ratio compared to zlib, a widely used compression library. However, the BQ-Tree is not optimized for processing large-scale geospatial data. To fill this gap, in this thesis, we propose two parallel BQ-Tree algorithms to compress and index large-scale geospatial raster data on General Purpose Graphics Processing Units (GPGPUs), called Multi-Block per Tile (MBPT) and One-Block per Tile (OBPT).

The key difference between MBPT and OBPT lies in the way they handle resource allocation. GPGPUs achieve parallelization by using hundreds of streaming processors organized within streaming multi-processors (SMPs) to perform tasks simultaneously. Both the proposed algorithms process raster data that is first partitioned into smaller parts

(tiles). MBPT partitions the tile into smaller areas (sub-tiles) that are mapped to different SMPs; the compression process for each sub-tile is executed in parallel within each SMP but requires synchronizing the sub-tiles results to get a unified compressed tile; this algorithm is designed to maximize the compression time of a single tile, thus it is ideal for large tile sizes. On the other hand, OBPT transfers the whole image to GPGPU memory and distributes tiles to different SMPs allowing the concurrent compression of many tiles at once. With this approach, each SMP will compress a tile by iterating in a sequential manner over the raster data, but within each iteration, the compression will be effectively parallel. This algorithm is designed to increase the overall raster data compression time by maximizing the number of tiles being compressed at once; it is advantageous if the raster data is divided into small but multiple tiles.

To study the performance of the proposed algorithms, we conducted extensive experiments comparing them with the multi-core BQ-Tree algorithm as well as a state-of-the-art geospatial parallel GPGPU compression algorithm, HFPaC, using the real dataset of satellite images (NASA MODIS). Our experiments show that compared with the multi-core BQ-Tree algorithm, OBPT achieves a compression time speedup between 3X and 9X while MBPT achieves a speedup of 5X on large tile sizes; and compared with HFPaC, the best performing proposed algorithm, OBPT, achieves a speedup of up to 2X for compression time and 2.5X for compression ratio, and yields a comparable average spatial query response time.

Chapter 1: Introduction

1. Objectives and Motivation

Geospatial data refers to data that is defined spatially by four dimensions: three of them correspond 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. Indeed, geospatial raster data is used in GIS applications

to perform analyses such as measuring temperature changes over time of a certain geographic area[62], monitoring deforestation[63], or land use over an area [64] using spatial queries. Furthermore, a compression technique that can also facilitate query processing is highly desirable. 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]), their high availability, and 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 GPUs. GPUs that are CUDA-capable are part of a generation of GPUs conventionally called General Purpose Graphical Processing Units (GPGPU) which support computations for applications that are traditionally handled on CPUs [61]. 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], 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. However, 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 thesis we aim to fill this gap by proposing parallel compression and indexing algorithms for geospatial raster data on GPGPUs.

The primary objective of this research is to develop parallel algorithms which not only efficiently reduces the size of very large geospatial data, but also can be used to answer spatial range queries. To achieve this objective, we perform the following tasks:

1. *Develop two parallel algorithms, OBPT and MBPT, for compressing and indexing geospatial raster data.*
2. *Run comprehensive experiments to measure the performance of the proposed algorithms compared to a multi-core version of the algorithm and a state of the art geospatial parallel compression technique.*

2. Background of Geospatial Raster Data Compression

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 has been a necessity.

By definition, data compression is the science of representing information in a compact form, which can be achieved by identifying and using structures that exist in the data [14].

As more diverse and sophisticated data such as audio files and images became available, the data compression field has grown to include techniques specifically targeting certain types of data. This was done either by modifying existing techniques to accommodate new types of data or by creating entirely new techniques based on principles learned from other scientific fields [15]. Different types of data present different challenges based on not only the inherent structure of the data but also on the end goal of the compression. For example, text compression techniques are mainly concerned with representing data in a compressed format that is completely lossless, otherwise the text would be incomprehensible; on the other hand, most image compression techniques deliberately reduce accuracy in order to decrease images size since small discrepancies in the visual representation of images are not easily detected by the human eye [16].

Most geospatial raster data is represented as images, and as a result, image compression techniques have been extensively used on geospatial data [17]; however because GIS applications are interested in drawing useful information from data, state-of-the art compression methods for geospatial data have be able to facilitate further analyses. As a result, image compression alone is not suitable. 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. The next section explores further the main issues relating to geospatial data compression.

3. Overview of Raster image compression issues

Compression ratio: The compression ratio is the ratio of the size of the uncompressed data to that of the compressed data. Every data compression algorithm is concerned with increasing the compression ratio. For example, reducing data to half of its size means that the storage capacity of the device is doubled. Although the cost of storage is increasingly cheap, the size of available data is growing fast due to technological progress; for example, geospatial image data has a higher resolution, is multi-spectral and is significantly larger than regular raster images [18]. Therefore, even though cheap storage is available, we are dealing with larger data and the need to reduce its storage requirements is still a pressing issue. Obviously, the compression ratio should be maximized given the sheer size of geospatial data, but other factors such as maintaining data accuracy limit the efficiency of compression. For example, most compression methods that were developed are lossy and relied on properties of the human visual system [19].These methods typically achieve higher compression ratios but at the expense of accuracy, which is often unacceptable for GIS applications that need to perform precise analyses on the data.

Compression Speed: As much as a compression algorithm can produce a high compression ratio, it should be able to run within a reasonable time. For some applications, trading off a high compression ratio for a fast compression is necessary. This issue is closely tied to the size of the data since the larger the dataset, the slower the compression. With the size of available geospatial data skyrocketing (for example, NASA

produces one terabyte of imagery per day [20]) compression techniques are required to reduce the data size in a reasonable time and be able to scale to increasingly larger data.

Fast Spatial Access: Fast Access refers to accessing an image fragment in the compressed file without having to retrieve the whole image. Although the purpose of compression is mainly to reduce the storage needs, it is becoming increasingly important that it allows fast processing. Typically when an image is accessed, the whole file is decompressed. However, with the increasing size of images, main memory size and I/O transfer are limiting factors; therefore, being able to access only targeted areas of an image is desirable. Techniques such as tiling, which is dividing the whole image into smaller regions (tiles), have been devised to allow access to targeted regions of the image [65]. Fast spatial access is particularly important in increasing the performance of spatial queries used during GIS analyses.

Spatial Locality: Maintaining spatial locality, meaning that neighboring objects in logical space are stored as neighbors on disk as well, is desirable when objects within close locations on disk are frequently accessed. Spatial locality allows the use of optimization techniques such as loading neighboring objects from CPU memory to on-chip caches thus reducing the time and energy for accesses [21]. Spatial queries, which are used by many GIS systems, are based on location or spatial relationships among objects. As a result, spatial locality among objects can increase the performance of these queries by reducing the I/O cost incurred while loading data from disk to CPU main memory and from CPU main memory to CPU caches, or from the CPU memory to GPGPU global memory.

4. Contribution

Geospatial raster data compression is important given the increasingly large scale and high resolution data available. Unfortunately most compression techniques do not take into consideration all the issues to be addressed for compressing large-scale geospatial raster data. We propose two algorithms, Multi-block per Tile (MBPT) and One block per Tile (OBPT) which uses GPGPU resources differently. Both of these algorithms operate on two-dimensional geospatial data partitioned into smaller parts called tiles. They both use the BQ-Tree (Binned-Quadrant Tree), which is a cache-conscious indexing data structure that has been proven to address three of the issues affecting the compression of geospatial raster data. The proposed parallel algorithms improve the BQ-Tree to address all four compression issues.

The first algorithm, MBPT, is a parallel algorithm which aim to maximize the use of GPGPU resources on a single partition (tile) of the whole geospatial raster data. This algorithm maximizes the compression time of a single partition at a time. The compression is repeated in a sequential manner for all the tiles of the geospatial raster data. The second algorithm, OBPT, aims to maximize the use of GPGPU resources on the whole geospatial raster data. It distributes GPGPU resources across multiple tiles of the geospatial raster data.

To the best of our knowledge, this is the first work that considers all the compression issues affecting geospatial raster data. We propose two novel parallel algorithms for compressing geospatial raster data on GPGPUs.

5. Organization

The rest of the thesis is organized as follows: Chapter 2 discusses the work related to data compression. It discusses existing compression techniques for geospatial raster data first, then it discusses available parallel compression algorithms. Chapter 3 describes in detail the data structure uses. Chapter 4 presents a thorough description of the proposed algorithms. Chapter 5 presents the experimental performance evaluations of the proposed algorithm. Chapter 6 provides a conclusion with a brief outline of each technique as well as their respective results. It also discusses future work for geospatial raster data compression and indexing on GPGPUs.

Chapter 2: Literature Review

1. Introduction

In this chapter we first discuss popular data compression methods used for geospatial raster data and analyze their advantages and disadvantages. In the second part of this chapter, we discuss available parallel algorithms of previously discussed techniques. The discussed compression techniques are classified in these categories: Entropy encoding, semantic-dependent, dictionary-based, transform based and compressed index techniques.

2. Entropy Encoding Methods

Entropy encoding compression methods refer to compression methods use code words to represent the original data in such a way that frequently occurring words will have a short code. For example, if text data is being compressed, a frequently occurring word such as e will be represented with bits "1" and a less occurring word such as z will have a longer code such as "000101". These compression methods are also completely reversible, meaning that the decoding algorithm is simply the encoding algorithm reversed, and can recover the original input data fully [61]. Entropy encoding compression techniques were among the first developed in the field of data compression [24] and they are the foundation of more complex techniques [23]. They are often used in conjunction with other compression methods.

2.1 Huffman Coding

Huffman coding is one of the most popular compression methods [15] and it has been used in many popular compression libraries such as GZIP [23] or image formats such as

JPEG[23]. Huffman coding is typically performed with two operations, the first step calculates the probability of data items based on how many times it appears in the data and assigns a symbol based on this probability. The data items probabilities are then used in the second step, which is the construction of a binary tree where the leaves represent each unique item in the data.

Assuming that an initial traversal of the data has been performed and each item in the data has been assigned a probability, the Huffman coding algorithm builds the tree in this manner:

- At the beginning, we have a collection of trees, each containing a single unique item from the data.
- The algorithm chooses two trees with the lowest probabilities, and build a new tree with the parent node having the sum of the probabilities of both items. Add the items as leaves and remove their individual trees.
- Repeat the process above until there is only one tree left.

At the end of this process we will have a tree with the leaves representing each symbol in the data. Once the tree is built, it has to be made available to the decoder. Since the path to each leaf is unique, the original data can be represented as a binary string where each symbol is represented by a sequence of bits describing the path to its location in the tree. Figure 1 below [63] shows data items already encoded in a tree based on their probabilities. For example with an original input of: AAAAEDCA, each value as an 8-bit data type, the string will use 64 bits. Now, when this input is compressed using the Huffman encoding tree below, it becomes: 00001111101010 which uses in total 14 bits.

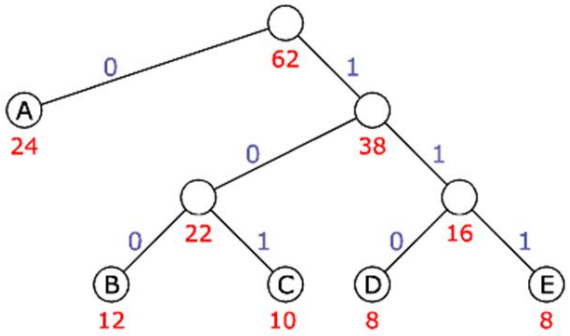

Figure 1: Huffman Encoding Tree

2.2 Arithmetic Coding

Arithmetic coding is considered to be an improvement over Huffman coding. Just like Huffman coding, it uses the probability of an item in the input data for encoding the item. Unlike Huffman coding, arithmetic coding does not use a give each item of the data a separate binary code, in fact the whole string can be represented as one single number [15]. Arithmetic coding however, involves more computation than Huffman coding.

Arithmetic coding is implemented in several steps as well. The first step assigns a probability p to each item. At the beginning of the compression, the interval has a range from 0 to 1 (represented as [0, 1)) and the probability p of each item is known. Then the data is read sequentially, and the interval is reduced based on the probabilities of each item read. To better understand this concept, we will use an example, as proposed by Solomon [15].

From Figure 2 [64], assuming we have three items A, B, and C with probabilities P1 = 0.5, P2 = 0.33 and P3 = 0.17, respectively; we assign to each of these items a range proportional to its size (the order in which these ranges are assigned is arbitrary). So A will have the subrange [0, 0.5), B will have subrange [0.5, 0.83), and C will have [0.83, 1.0). Given an input of BCA, the compression follows these steps (illustrated in Figure 2).

The first item B of the input is read, the initial range [0, 1) is reduced to [0.5, 0.83), and the binary number 0.10x is the shortest code that can represent an interval in range [0.5, 0.83). The shortest binary code for floating point values in this range is 0.10 (10 is 2 the denominator, so the floating point number would be 1/2 = 0.50).

The second item C is read and the interval is reduced to [0.73, 0.84) through these calculations which shrink both the upper and lower limit of the previous range. The shortest binary code for floating point values in this range is 0.11001 (11001 is 2 the denominator, so the floating point number would be 1/2 = 0.50).

Lower limit: 0.5 + (0.83-0.5)*0.83 = 0.7739

Upper limit: 0.5 + (0.83-0.5)*1 = 0.83

The last symbol A has an interval of [0, 0.5) when applied to the previous interval gives [0.7739, 0.801).

Lower limit: 0.7739 + (0.83-0.7739)*0 = 0.7739

Upper limit: 0.7739 + (0.83-0.7739)*0.5 = 0.801

After the whole data is read, any number within that range is chosen to represent the input.

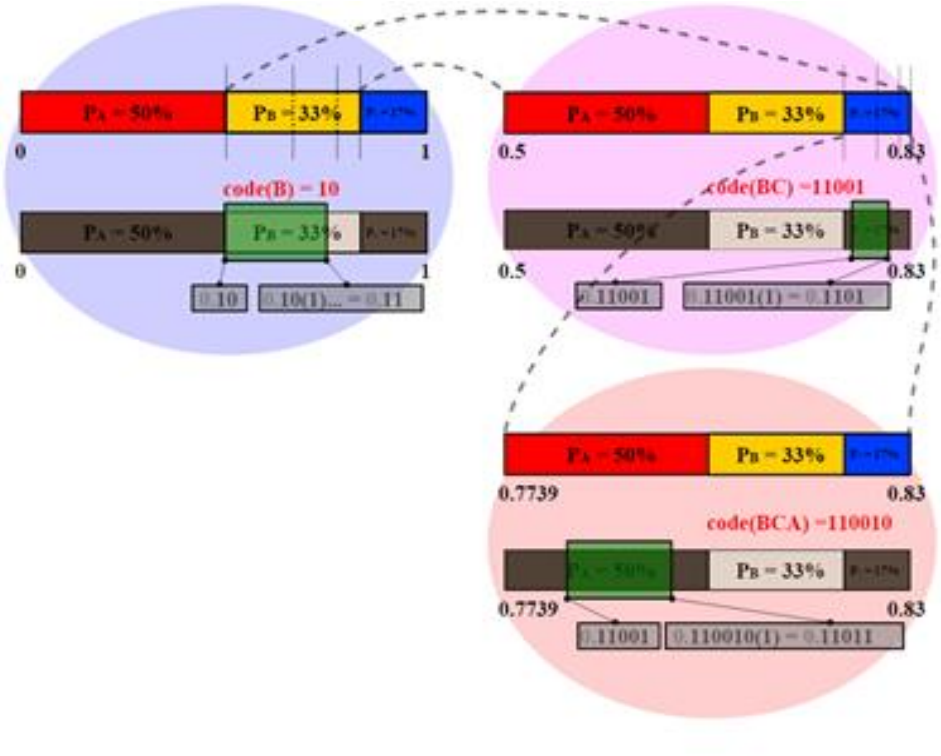

Figure 2: Illustration of Arithmetic coding

To decode the compressed format, the decoding algorithm has access to the probabilities of each of the items and the original data size. It can effectively deduce the order in which the items were ordered in the original input, and as such it is able to recover the original input data.

2.3 Advantages and Advantages

Entropy encoding techniques under this category focus on maximizing the compression ratio above all else. They are also independent of the type of input data provided which has made them widely used in data compression. However, they fail to take into account spatial parameters which as we have discussed are important to geospatial data. Classic implementations of entropy encoding methods do not take advantage of the spatial properties of geospatial data and thus do not preserve spatial locality. Furthermore, they

fail to facilitate fast spatial access on geospatial data because accessing an item requires that all the previous items get decompressed first.

Finally, entropy encoding methods are computationally intensive because of a number of factors, first the probabilities of each of the items in the data has to be calculated, which incurs a time complexity of O(n), and in the second step, the same data is again sequentially read in O(n) time. Clearly, for very large data sizes, these techniques are not easily scalable thus the compression time increases.

3. Semantic-dependent Methods

Semantic-dependent methods are designed to take advantage of local redundancy in the input data. They are mostly used with image compression because images often present large areas of local redundancy. Popular semantic-dependent techniques (techniques that take into account the layout of the data), which have been widely studied ([24] [25] [26]), such as run-length encoding and difference mapping, rely on the premise that most images have high local redundancy [25].

3.1 Run-Length Encoding

Run-length encoding compresses the data by encoding the number of times an item is successively repeated in the input. For example if the data is BBBBWWWWW, the run-length encoding algorithm will store B4W5 where 4 and 5 represent the number of times B and W are repeated respectively [27].

3.2 Differential Mapping

Differential mapping compresses the data by giving a code to an item based on its relationship with the previous item [66]. For example, with an array of values [1001, 1002, 1003, 3000, 500], the differential mapping algorithm will store the encoded version

as [1000, 1, 1, 1997, -2500] [28]. This method is useful if the items in the data are either highly locally redundant or the items value change slowly.

3.3 Advantages and Disadvantages

These methods are simple to implement and do not require any preliminary statistical analyses. For highly redundant data, the space reduction can be significant. However these methods, in their classic implementations, are not scalable to larger data because of the sequential nature of the process, thus the compression time will be a limitation. Also fast spatial access is not facilitated because accessing any item in the dataset requires the decompression of all the previous items.

4. Dictionary-based Methods

The previously described methods assume independent symbols that are not correlated to each other. Dictionary based methods, on the other hand, assume that some symbols are highly correlated and will most likely occur often in the same pattern in the data. As such, these patterns along with a representative code are stored in a list or a dictionary. The patterns in the data are then replaced by their representative code, which is more compact. The dictionary is shared with the decoder and will be used to recover the original data [27]. Dictionary-based methods are mostly used for text compression, where a repetition of patterns is expected, however it is used for spatial data as well [29][30]. The most significant of dictionary methods is the Lempel-Ziv Encoding technique(L7ZZ) and it has been the basis of different variants such as LZW (Lempel Ziv Welch encoding).

4.1 Lempel-Ziv Encoding (LZ77)

LZ77 is one of the most popular dictionary-based compression methods [67]. LZ77 is the basis for popular image formats such as PNG, GIF, and ZIP. LZ77 builds an adaptive

dictionary, meaning that the dictionary changes as the data is encoded. The LZ77 technique maintains a sliding window divided into two parts (Table 1), a search window and a look-ahead window. The search window is a fixed-size window containing patterns recently encountered and as the window slides, the search window's contents change. The look-ahead window contains the next items to be encoded. As the window slides, the items in the look-ahead window are encoded into tokens based on the contents of the search window. If an item cannot be effectively encoded based on the current search window, then the search window is incremented to accommodate the new item. These tokens contain three values: the offset to the match in the search window, how big the match is, and the next character to be encoded.

Table 1: LZ77 applied to a string

Search window	Look-ahead window	Tokens
	sir*sid*Steadman*	[0,0,"s"]
s	ir*sid*Steadman*	[0,0, "i"]
si	r*sid*Steadman*	[0,0, "r"]
sir	*sid*Steadman*	[0,0, "*"]
sir*	sid*Steadman*	[4,2, "d"]

4.2 Advantages and Disadvantages

Dictionary-based methods are highly performing on input data with long and highly repetitive patterns such that the resulting compressed format will contain representative codes that are significantly smaller than the actual pattern. However, dictionary-based methods do not facilitate fast spatial access because each compressed item depends on all the previously compressed items.

5. Transform-based Techniques

Transform-based compression techniques first transform the original data such that the subsequent compression will be more efficient. For example, before applying run-length encoding, the data can be rearranged such that similar items are grouped together thus increasing the compression efficiency [62].

5.1 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is usually applied on text compression. The BWT takes a block of data and rearranges it using a sorting algorithm. The resulting output block contains exactly the same data elements that it started with, differing only their ordering [31]. Unlike previously described methods, the BWT aims to process large blocks of data together at once. As shown in Figure 3, the first step of the BWT creates N permutations of the original string of size N. Secondly, as shown in Figure 4, the permutations, represented as a matrix are sorted such that repeating first characters are adjacent. From example, Row 3 from Figure 3 is moved below Row 2 in Figure 4 since they both begin with a C.

Row 0	A	C	B	F	C
Row 1	C	B	F	C	A
Row 2	B	F	C	A	C
Row 3	F	C	A	C	B
Row 4	C	A	C	B	F

Figure 3: Permutation step of BWT

Row 0	A	C	B	F	C
Row 1	C	B	F	C	A
Row 4	C	A	C	B	F
Row 2	B	F	C	A	C
Row 3	F	C	A	C	B

Figure 4: Sorting step of BWT

At the end of the process, the BWT output will be the last column of the sorted matrix of permutations (CAFCB) as well as an integer indicating the string containing the first occurrence of the first character in the sorted matrix (in our case the first character is A, and the first row containing A is row 0). Therefore with string CAFCB and integer 0, we can perform the permutations in reverse and get the original data.

Clearly, the BWT in itself does not perform any compression of the data. However, it rearranges the data such that if a semantic-dependent compression technique such as run-length encoding is applied, the compression ratio achieved will be increased.

5.2 Other Transform-Based Methods

Most transform techniques are usually used for the compression of images. They are essentially lossy techniques but because of their wide use in the image compression field, it is important to provide a brief overview. As explained in [32], wavelet transform methods are based on decomposing the image into multiple layers using different resolutions. Wavelet transform is appealing because it can estimate when certain values appeared in the signal at a certain scale, and this scale can be thought of as the frequency of the signal. Wavelet transform methods therefore can be used to approximate where and how often a certain value appears in an image. For each wavelet transform scale, each

pixel in an image is represented with a coefficient. These coefficients are then easily compressible because they are usually concentrated in a smaller range than the original values.

Another popular transform method is the discrete cosine transform which expresses data points as a sum of cosine functions oscillating at different frequencies. Afterwards, pixels occurring only at non-important frequencies are discarded in order to reduce the size of the data (thus the "lossy" aspect of this method) through a process known as quantization [33]. The final remaining coefficients are then compressed using any of the compression techniques described in the previous sections.

5.3 Advantages and Disadvantages

The BWT involves heavy computations for rotating and sorting blocks of data. It focuses on achieving a high compression ratio by rearranging the data at the expense of time efficiency. Additionally, the BWT requires the compressed format to be first decompressed in order to perform any further spatial analyses. Therefore it does not facilitate fast spatial access.

Other transform methods are essentially lossy and although they achieve a high compression ratio, the decompressed data contains a certain level of error and cannot be effectively used for rigorous data analyses.

6. Compressed Index Techniques

To understand compressed index techniques, it is important to first define what an index is. An index is a way of representing data in a way that makes it easy to look up information. Depending on the user end-goal, an index can be an inverted index [34]

meaning that the index contains all the unique values in the original data and where they appear. Inverted indexes are widely used in the field of information retrieval to facilitate query processing. Indeed, instead of scanning a document for a specific term, the term location can be easily looked up in an inverted index. Inverted indexes can be extended to include how many times a specific value appears in the dataset. An index can be a bitmap index, where the index is essentially a two dimensional array B with one column representing an attribute and each row representing an item in the original data (more bitmaps are generated if more than one attributes exist); all the queries are solved using bitwise operations mainly. If an item at row r has an attribute represented at column c, the bitmap location B[r] [c] will contain bit 1, and 0 otherwise. Clearly complex indexes facilitate query processing but they are not space efficient. The more elaborate an inverted index is, the faster query responses will be [34]. Indexes can be compressed by encoding them in a space efficient manner

6.1 Word Aligned Hybrid

The Word Aligned Hybrid (WAH) is a compression technique that operates on inverted indexes [55]. The purpose of the WAH technique is to facilitate query processing while maintaining a good compression ratio. The WAH applies compression at the bit-level and byte-level (thus the use of hybrid in its name). A popular way of representing an inverted list for a data item is to apply differential mapping on the list (described in Section 2.2). For example, if an inverted list for one item in the data is: <4, 10, 11, 12, 15, 20, 21, 28, 29, 42, 62, 63, 75, 95>, then it becomes <4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12, 20>. Clearly this process reduces the number of bits of each number; with this observation, it is apparent that each number can be represented using fewer bits. Furthermore, all the

unused most significant bits are removed and the remaining bits are packed into 32-bit words, occupying 28 bits while the first 4 bits are used as a flag. The purpose of the flag is to indicate how many values were compacted in that single word. Indeed the WAH assigns before-hand how many bits each 32-bit word will contain by analyzing contiguous values. For example, if the remaining values are all 1s, then 28 of them can fit in one 32-bit word (remember the remaining 4 bits are reserved for a flag), if the remaining values use only 2 bits, then 14 values can fit in one 32-bit word (2 bits per value), etc…For example in the example above, first checks if it can fit the whole list in one word, and that is impossible since some values use more than one bit; it checks again the first 14 bits to see if they only use 2 bits and can fit in a 32-bit word, then the algorithm checks again if the first 8 values use only 3 bits, which is correct, so the first 8 values are stored in one 32-bit word (1 is represented as 0, so each bit representation is 1 less than the original value): <011, 101, 000, 000, 10, 100, 110, 000>. The flag will then have a code representing "8" as the number of values contained in the word.

Decompression is fast because the decoder can just look at the flag at the beginning of each word to know how many values to generate from the 32-bit word. In the example above the 32-bit word will be <"8", 011, 101, 0, 0, 10, 100, 110, 0> and it will be decompressed as (assuming the values occupied 8 bits originally) <00000011, 00000101, 00000000, 00000000, 00000010, 00000100, 000000110, 00000000>, then 1 is re-added to offset the values into their correct initial amount < 00000010, 00000110, 00000001, 00000001, 00000011, 00000101, 00000001>.

6.2 Advantages and Disadvantages

Compressed Index Techniques are appealing because their end goal is not just achieving a good compression ratio but also to allow fast access of data items. However they do not take into account the spatial aspect of data, therefore they do not facilitate spatial query processing. Also, compressing requires going over the same chunks of data several times, which makes it unsuitable for large data.

7. Spatial Indexing Methods

In this section we present popular hierarchical data structures used for indexing to facilitate querying, namely the R-Tree and the BQ-Tree. Although the R-Tree indexing data structure does not compress the original data, it is relevant because it aims to maintain data locality and fast access for spatial queries. The BQ-Tree is reviewed because it is not only an indexing data structure but it can also be used for compression.

7.1 The R-Tree

The R-Tree is a spatial data structure widely used for indexing in the spatial data processing field [45]. The R-Tree was developed in order to handle multi-dimensional data by taking into account spatial locality and fast spatial access. It is implemented by recursively dividing the original spatial data into smaller regions. Each node of the R-Tree contains two fields: a minimum bounding box, which is the rectangular region that fully encloses a certain region r of the spatial data, and the location of the region in the database [45]. The leaf nodes of the R-Tree contain the minimum bounding boxes of individual points, segments or areas of the original data. For example, from Figure 5, leaf node a, has a bounding box enclosing region 1 and 2, and parent node A, has a bigger bounding box covering both node a and b. Indexing spatial data with this technique

ensures that spatial queries such as spatial range queries or nearest neighbor queries are processed faster. Indeed, instead of loading in memory the whole spatial data and applying the spatial query, the R-Tree is first traversed up to the tree node it is interested in. For example, from Figure 5 [65], if a spatial query is asking for a region enclosing areas 3 and 4, the R-Tree is traversed starting at the root node "root", the bounding boxes of its children are checked to see if the area being searched for is within the boundaries of any of the two children. In this case, child node A is chosen. From node A, the process is repeated and the traversal completes and returns node "b" containing areas 3 and 4.

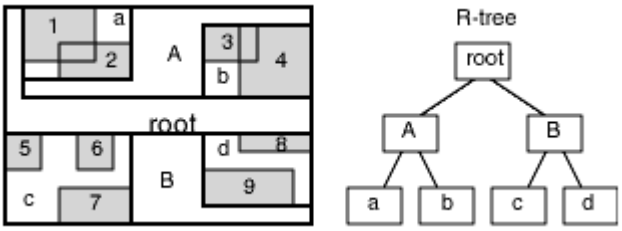

Figure 5: The R-Tree

7.2 The Quadtree

Given a finite set of points in a 2-dimensional space, the quadtree data structure 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 [35], but we focus on *region quadtrees* which can be used to represent binary images, multicolored data, or grayscale images [36].

From Figure 6 [67], a bitmap composed of 1s and 0s will be represented as a tree with a branching factor of four, where black leaf nodes represent regions of the raster (referred to as quadrants) composed of only 0s, white leaf nodes represent quadrants composed of only 1s, and gray nodes represent internal nodes (the ordering of node is in z-order [67]).

From Figure 6, a spatial query asking for nodes in region with (x,y) boundaries from (4,5) to (4,6) starts by traversing the tree down to the smallest node containing both coordinates (4,5) and coordinate (5,6), which in this case is level 3. Once on that level, the area between the two boundaries is effectively contained in the subtree, from our example, there is no further traversal needed because the smallest node enclosing the boundaries is a black node, which implies that the whole region is composed of black nodes.

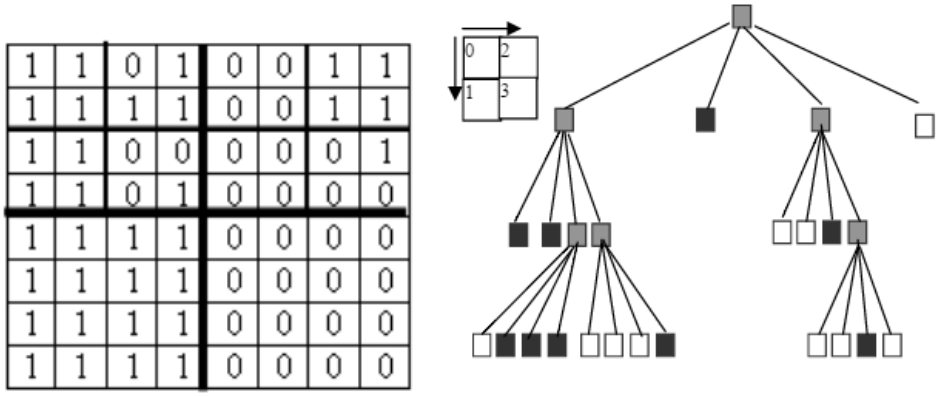

Figure 6: A Region Quadtree

7.3 Advantages and disadvantages

Hierarchical indexing techniques are appealing because they facilitate spatial locality and fast spatial access. They minimize the I/O cost of queries by reducing the amount of data to be loaded in main memory for query evaluations. However, hierarchical indexing methods are not space-efficient in their classic implementations. Some research efforts have been done to reduce the size of the indexes [48] [49], but the original data on disk is still uncompressed. The classic R_Tree and quadtree are pointer based implementations 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, therefore storing pointers can potentially be inefficient for storage for large-scale data.

Additionally, building indexing structures 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.

8. Floating-point Compression Techniques

Floating-point compression techniques deserve to be mentioned because they are optimized for processing large-scale data. The methods presented below are notable because they highlight innovative ways of improving compression by analyzing the byte composition of data values.

8.1 ISOBAR Pre-conditioning

The ISOBAR pre-conditioner [50] 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 [55] or bzlib2). The ISOBAR pre-conditioner categorize chunks of data as incompressible or compressible using a threshold that is either provided by the user or calculated through data sampling. The ISOBAR pre-conditioner follows the steps below:

1. Given a dataset with N values each with b bytes, divide it into b chunks of data each with N values of 1 bytes.
2. For each data chunk, calculate the frequency of each value and if any of the values is above or below a certain frequency, qualify it as compressible or incompressible.

3. Combine together data chunks labeled as compressible and compress them with either zlib or bzlib2.
4. Store the metadata indicating which data chunks were compressed to allow correct decompression.
5. Significant improvements on the compression ratio were observed on highly random scientific data [50]. The reason being that by dividing the data into bytes, more uniform data distribution is observed which works very well with LZ77-based compression libraries such as zlib and bzlib2. Additionally, these compression methods work better at the byte level, therefore finding ways to reduce the size of individual data items to be processed improves the compression speed [50]. Up to 40% compression ratio improvement was observed over using just zlib or bzlib2. And the compression speed was increased up to 120X. Additionally, the ISOBAR pre-conditioner facilitates fast spatial access at the chunk level is preserved because each of them can be individually decompressed. However, pre-conditioning data first increases the time required to perform the compression especially if dealing with large-scale data. Also, fast spatial access is not preserved inside each chunk. Indeed, because zlib and bzlib2 use a combination of LZ77, Huffman coding (for zlib) and BWT (for bzlib2) and as described in Sections 1.3,3.2, and 4.3, these techniques do not facilitate fast spatial access.

8.2 FPC

FPC, a technique developed in [57] losslessly compresses 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 such as XOR and bit-masking. The FPC performs the prediction using variants of two popular table-based value predictors. The FPC predicts the next value in a dataset by using the two value predictors on the previous value, then the closest to the actual value is chosen, then an XOR operation is performed on both the predicted and actual values, and finally the resulting bits are stored. Additionally, a bit flag is stored to indicate which value predictor should be used during decompression. Decompression operates in the exact same way because XOR is reversible.

8.3 Advantages and Disadvantages

Significant improvements on the compression ratio were observed on highly random scientific data using ISOBAR (described in Section 7.1). The reason being that by dividing the data into bytes, more uniform data distribution is observed which works very well with LZ77-based compression libraries such as zlib and bzlib2. Additionally, these compression methods work better at the byte level, therefore finding ways to reduce the size of individual data items to be processed improves the compression speed [50]. Up to 40% compression ratio improvement was observed over using just zlib or bzlib2. And the compression speed was increased up to 120X. Additionally, the ISOBAR pre-conditioner facilitates fast spatial access at the chunk level is preserved because each of them can be individually decompressed. However, pre-conditioning data first increases the time required to perform the compression especially if dealing with large-scale data. Also, fast spatial access is not preserved inside each chunk. Indeed, because zlib and bzlib2 use a combination of LZ77, Huffman coding (for zlib) and BWT (for bzlib2) and as described in Sections 1.3,3.2, and 4.3, these techniques do not facilitate fast spatial access. FPC

(Section 7.2) achieves a higher compression ratio on randomly distributed datasets compared with gzip (up to 15 times higher). Additionally, it achieves a higher compression speed compared to the aforementioned compression techniques (up to 5X higher throughput) [50].

9. Compression Techniques on GPGPUs

In the previous section we gave an overview of compression techniques for compressing geospatial raster data on CPUs. In the following section we present available parallel versions of these algorithms on GPGPUs.

9.1 Brief GPGPU Overview

Before describing parallel compression algorithms, it is important to understand in broad terms how GPGPUs work. GPGPUs contain thousands of lightweight threads which perform simple operations in parallel. Threads are organized in thread blocks and within each block they are further subdivided into warps. Threads are subdivided into subgroups of 32 threads each referred to as a warp; and threads within a wrap execute a single instruction in lockstep.

9.2 Parallel LZSS

LZSS is a derivative of LZ77 where the code generated to represent each pattern of items (Refer to Section 3.1 for the LZ77 description) is guaranteed to be smaller [53]. This technique first checks the size of the generated code and decides if its length is small enough to be stored, otherwise the original pattern is used in the compressed format. Two parallel implementations of LZSS were proposed in [54], each one using a different resource allocation strategy. As illustrated in Figure 7,**Error! Reference source not**

found. on the left side labeled “version 1”,The first naïve version divides the dataset into smaller chunks and each one is allocated to a single GPGPU block (labeled block 0 to block N) which in turn divides the chunks into smaller parts that are assigned to threads and processed in parallel. The compressed chunks from each block are combined sequentially on the CPU. On the right-hand side of **Error! Reference source not found.** abeled “version 2”, the second approach introduces a deeper parallelization level. As described in Section 3.1, an item is read and the LZSS algorithm looks into the recently visited items (referred to as the search buffer) to check if there is a match. This process of finding a match in the visited items is parallelized in the second version of CULZSS by utilizing many threads, each one reading a single value from the search buffer The second approach which is the best approach performs 3 times better than the multi-core CPU implementation.

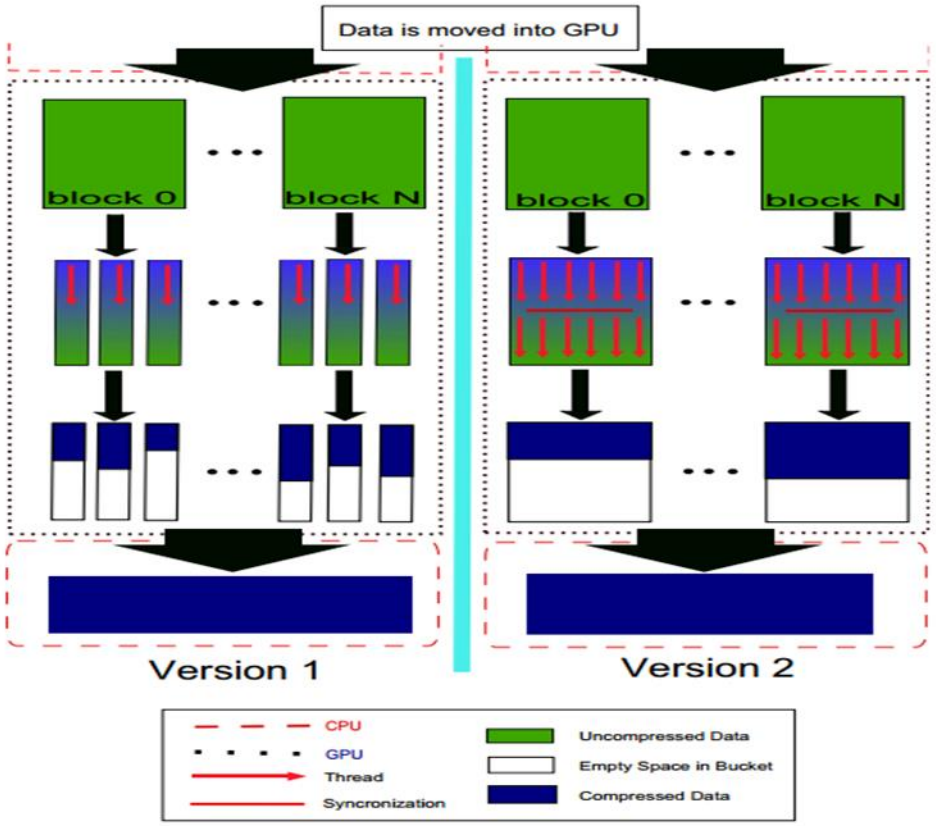

Figure 7: Both versions of CULZSS

This method achieves a high compression speed as well as a good compression ratio especially on text data [54]. However, one has to decompress all the data items preceding the item being searched, which does not encourage fast spatial access. Additionally, this method operates on one dimensional data, which does not facilitate spatial query processing.

9.3 HFPaC

HFPaC [51] is a type of compression algorithm for height field data. Height field data refers to a two-dimensional grid where each gird cell represents the height of a sampled point as shown in Figure 8.

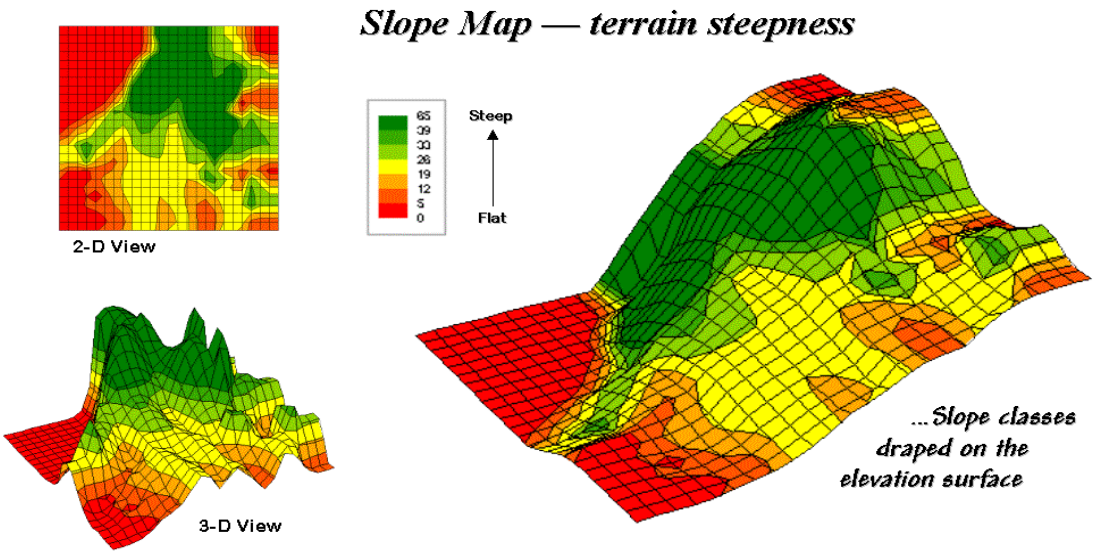

Figure 8: Example of Height Field data

HFPaC predicts the content of a cell by using Bezier surfaces. A Bezier surface is a type of mathematical function that use control points to draw a surface fully contained within these control points. As shown in Figure 9, 11 control points labeled from $C_{0,0}$ to $C_{2,2}$ are used in a Bezier function to draw the surface in yellow below the control points.

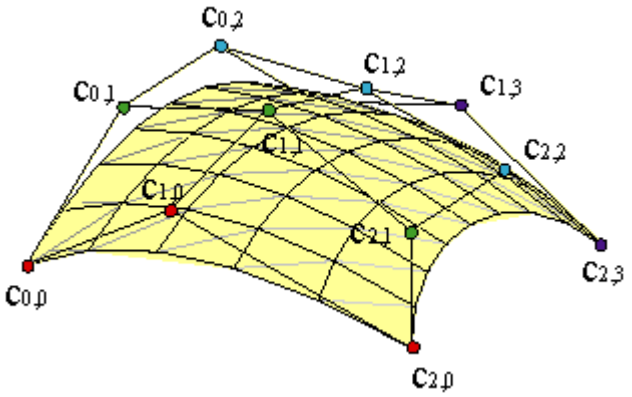

Figure 9: Bezier Surface

HFPaC first calculates Bezier control points based on the height field data and stores them in a layer. Then using Bezier surfaces, a grid is drawn based on the control points. Then, the approximated value of each grid cell is subtracted from the original value in the height field data. The error from the subtraction is stored in two other levels. Together, these three layers represent the compressed form of the original height field data (Figure 10).

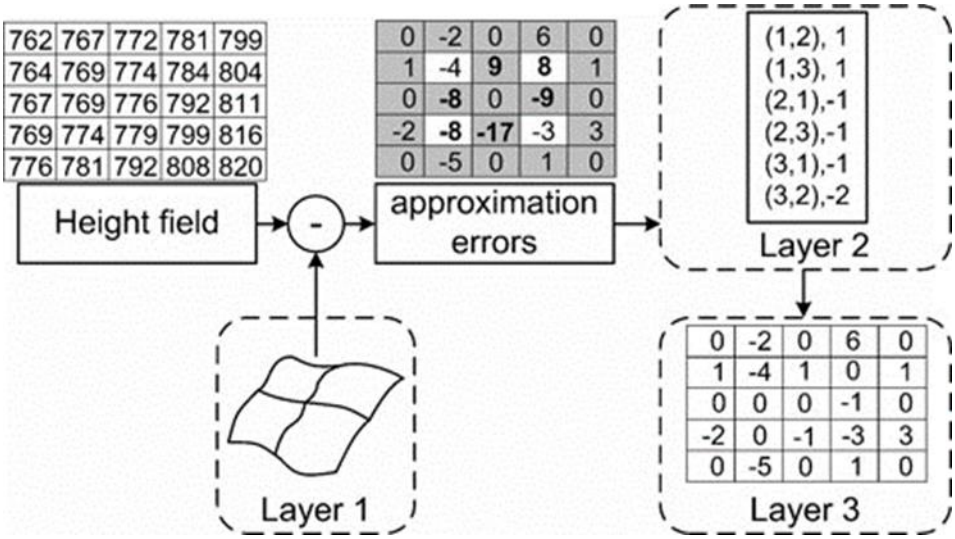

Figure 10: HFPaC compression technique

Approximating values and calculating the error is performed in parallel on CUDA processors. Each segment is allocated GPGPU resources which work simultaneously. As a result, this technique achieves a good compression speed due to the use of GPGPUs; it also maintains spatial locality by maintaining the two dimensional structure of the original data and it also supports spatial access because each data point in the compressed format can be decompressed and accessed individually. However, this technique only achieves a high compression ratio if the data is very smooth such as height field data, otherwise the approximation is very high compared to the original values and, as a result, the stored error is too high. The best GPGPU implementation reduces by half the

compression time compared to a single-core CPU implementation; and achieving a compression ratio comparable to another state of the art height field compression technique [51].

9.4 GPU-WAH

The GPU-WAH is an extension of WAH (described in Section 5.1) that allows to parallelize compressing and decompressing on GPGPUs [60]. GPU-WAH compression is executed in three major steps: bitmap extension and compression.

The first step, bitmap extension appends a single 0 bit after each 31 bits in the input bitmap B. Before appending the 0 bit, the algorithm first pads the input bitmap with as many 0 bits as necessary to make its size a multiple of 31. Then the number *n* of 31-bit sub-bitmaps is calculated. From the number *n* of 31-bit sub-bitmaps, the size of the output bitmap is calculated. Furthermore, each 31-bit sub-bitmap of B is copied to a specific location in E and 0 bit is appended at the end (effectively creating a 32-bit word). This part of copying each sub-bitmap of B in E is performed in parallel.

The second step is the actual compression which is performed in the following sequential stages (1-5), but each stage contains operations performed in parallel:

1. The class of each of the 32-bit words is determined and the class representing how bits are arranged within each word [60].
2. 32-bit words of the same class are put into blocks. An alternate array *F* of 1s and 0s of a size equal to the total number of words in the original bitmap is created. F records 0 at index *i* if the word at location *i* is at the end of its block or 1 if otherwise.

3. Using the array *F*, the number of blocks *b* is calculated in parallel; this is important because each block will be compressed into one 32-bit word and the number of blocks *b* corresponds to the final number of words after word compression is performed.
4. An array *T* is created containing the cumulative number of words from every block. This stage is performed in parallel as well.
5. The last stage reads in parallel the words contained in extended bitmap E for which array *F* recorded a 1. Each of these words gives an indication of which class the rest of the words in the block belongs to. From this information, each word within a block can be compressed in parallel using the maximum bits allocated for that particular class.

This algorithm achieves compression times which are 16X to 24X times faster than the single-core CPU algorithm. However, when the GPU transfer times are included, the improvement is only 3.6 to 5.8 times faster [60]. As mentioned in section 5.1, WAH does not address the spatial aspect of data, in fact, the way compression is designed, with words being rearranged within the original bitmap, applying spatial queries requires the decompression of the data first.

9.5 Floating Point Compression on GPUs

GFC [58], a floating-point compression optimized for GPUs uses a similar prediction mechanism as FPC (Section 7.2). GFC operates in the following steps:

1. The floating-point data to be compressed is divided into *n* chunks of 32 bits or multiples of 32 bits.

2. Each chunk is written into an integer multiple of 32 bits; if necessary padding is applied if the chunks are less than 32 bits. Each chunk is compressed independently.
3. Each chunk is divided into sub chunks of exactly 32 bits.
4. The GFC algorithm starts with the first sub chunk of 32 bits; it predicts a value based on a prediction table, then it subtracts the predicted value from the actual value of the sub chunk and the result is stored.
5. The residual from the step above is stored in a compressed format with 4 bits representing the sign of the residual, the number of leading zeros is stored in the next three bits, and then a variable size residual is stored (stripped of its leading zeroes).

The steps above are performed within a single *wrap*. Each warp works on one chunk and never communicates with other warps. Then the chunk is processed sub-chunk by sub-chunk, in a sequential manner within each wrap.

This implementation was four times faster than the same implementation on multi-core CPU. However, the compression ratios registered were lower than FPC compression ratio [58].

9.6 Parallel R-Tree

A parallel implementation of the R-Tree was proposed in [59]. The approach taken is to use a linear array instead of pointers to represent the R-Tree which facilitates data transfer to the GPGPU. Each non-leaf node of the 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. Two implementations of the R-Tree are proposed.

The first approached (referred to as the low-x packing approach) first sorts the original minimum bounding boxes (MBRs) using their x values. The second implementation first sorts the MBRs based on the x-axis, then the space is divided into slices and each slice is sorted according to the y-axis.

The first implementation follows the following steps:

1. The original MBRs are sorted along the x-axis.
2. Every *d* MBRs are packed together into the level above. By packing, it means that the parent MBR is calculated, the position of the first child (first of the d nodes) is stored and the length *len* contains *d*.
3. Step 1 and 2 are repeated until the root is created.

   The second implementation, the Sort-Tile- Recursive (STR), follows these steps:

   1. The original MBRs are sorted along the x-axis.
   2. The sorted MBRs are divided into slices of size *t* which is predefined.
   3. Every d MBRs in a slice are packed together into a parent node.
   4. Steps 1, 2 and 3 are repeated until the root node is created.

The parallel aspect of these algorithms is introduced during the creation of each level of the tree. GPGPU threads work together to create each level of the tree.

Up to a 4X speedup was observed over the same implementation multi-core CPU. The R-Tree constructed using STR provides the best results for increasing the performance of spatial queries [59]. However, as explained in Section 6.3, spatial indexing methods do not compress the data, therefore they are not fully competitive with other compression techniques.

10. Conclusion

In this chapter we first reviewed relevant compression algorithms on CPUs. Then we have reviewed the parallel versions available for GPGPUs. We highlighted the advantages and disadvantages of each family of algorithms in an effort to provide a better understanding of what an efficient compression algorithm for geospatial data requires and to showcase the lack of such an algorithm. Table 2 shows a summary of the features supported by each family of compression and indexing techniques. As evidenced in Table 2, entropy encoding, semantic-dependent, dictionary-based and transform-based algorithms aim to achieve a high compression ratio at the expense of compression time. Compressed index compression algorithms aim to strike a balance between a good compression ratio and fast spatial access through facilitating decompression. Indexing methods facilitate fast spatial access and maintain spatial locality but in their classic implementations, they do not compress the data. Based on Table 3, all the parallel algorithms reviewed support an efficient compression time and only HFPaC addresses all four geospatial raster data issues. However, HFPaC is optimized for height field data and is not guaranteed to perform well on any geospatial raster data. Finally, compression using BQ-Tree is a good candidate for solving all the compression issues; however its current multi-core implementation does not efficiently address the issue of compression time. In this thesis we will propose an improved algorithm using the BQ-Tree which will address the compression time issue through the use of parallelism.

Table 2: Comparison of compression algorithms based geospatial raster data compression issues

Compression/Indexing Techniques	Compression Ratio	Compression Time	Fast Spatial Access	Spatial Locality	Geospatial Raster Data?
Entropy Encoding	Yes	No	No	No	Yes
Semantic-Dependent	Yes	No	No	No	Yes
Dictionary-based	Yes	No	No	No	No
Transform-based	Yes	No	No	No	Geospatial images
Compressed Index	Yes	No	Yes	No	No
Spatial Indexing	No	No	Yes	Yes	Yes
Floating-point	Yes	Yes	No	No	No
BQ-Tree	**Yes**	**No**	**Yes**	**Yes**	**Yes**

Table 3: Comparison of parallel compression/indexing algorithms based on geospatial raster data issues

Compression/Indexing Techniques	Compression Ratio	Compression/Indexing Time	Fast Spatial Access	Spatial Locality	Geospatial Raster Data?
Parallel LZSS	Yes	Yes	No	No	No
HFPaC	**Yes**	**Yes**	**Yes**	**Yes**	**Height Field**
GPU-WAH	Yes	Yes	No	No	No
Parallel R-Tree	No	Yes	Yes	Yes	Yes

Chapter 3: The Proposed Compression Algorithms

1. Introduction

In this chapter we provide a definition of the data structure, BQ-Tree [36], used in our compression algorithms and its proof of losslessness. Then we point out the shortcomings of the sequential compression algorithm applied on the BQ-Tree and propose alternative parallel algorithms for GPGPUs. Before describing the BQ-Tree, here is a list of terms that will be used in the next sections.

Definitions

Bitmap: refer to a two-dimensional matrix of *mxn* size with each matrix cell containing a 16 bit value. A two dimensional geospatial raster data can be thought of as a matrix, therefore instead of using the word geospatial raster data, we will use bitmap.

Cell: refers to a value in the bitmap located at row *r* and column *c*.

Bitplane: refers to a single bit at position *i* within each bitmap cell value.

*Bitplane bitma*p: refers to a two-dimensional matrix of *mxn* size with each matrix cell containing a 1 bit value. A bitplane bitmap can be thought of as a component of the bitmap; where 16 bitplane bitmaps combined together will produce a singular 16bit bitmap.

Depth: refers to the number of bitplanes that a bitmap contains. Each *ith* bit contained in a bitmap cell value represents a bitplane at depth *d.*

2. The BQ-Tree

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 $2q \times 2q$ is chosen such that q≤m. BQ-trees 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 $2i \times 2^i$ 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 11 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 $2m\text{-}q \times 2m\text{-}q$ elements, there is a one-to-one correspondence between the last-level quadrants and the nodes in the last-level shown in Figure 12. 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 $2q \times 2q$ 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 11 where the last-level quadrants have size $2^1 \times 2^1$(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.

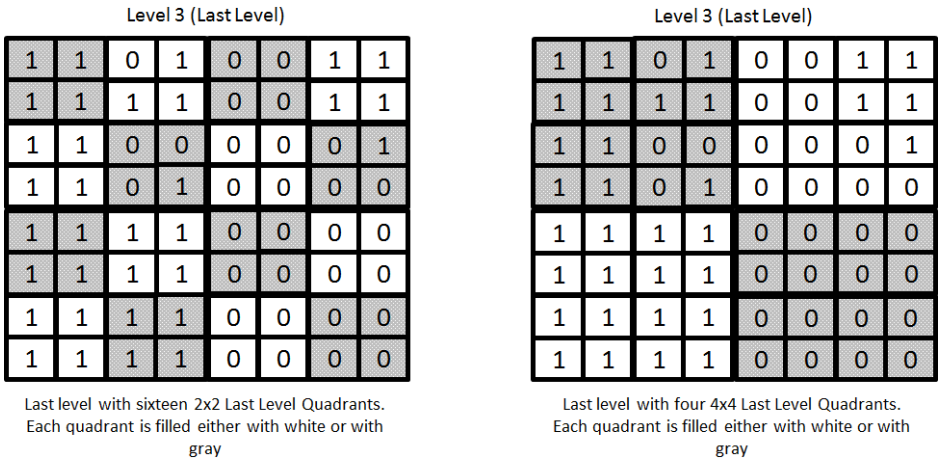

Figure 11: Choice of size for the Last-Level Quadrants (LLQS)

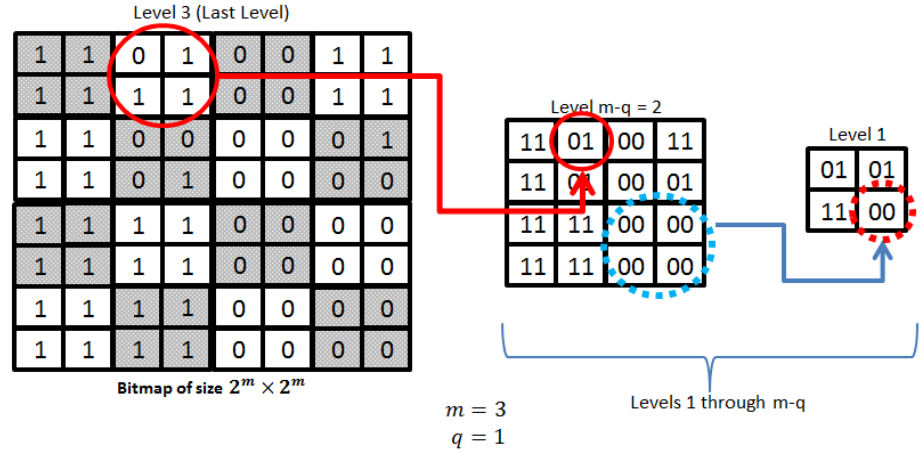

Figure 12: Levels of a BQ-Tree

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 12 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. BQ-Tree Compression

In this subsection, we explain how BQ-Trees are physically built. 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 [17]: 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 2m×2m and a last-level quadrant of size 2q×2q, 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 13, which shows the construction of the pyramid array, and Figure 9, which shows the construction of the LLQS array. In the following sections, we describe how these two arrays are constructed.

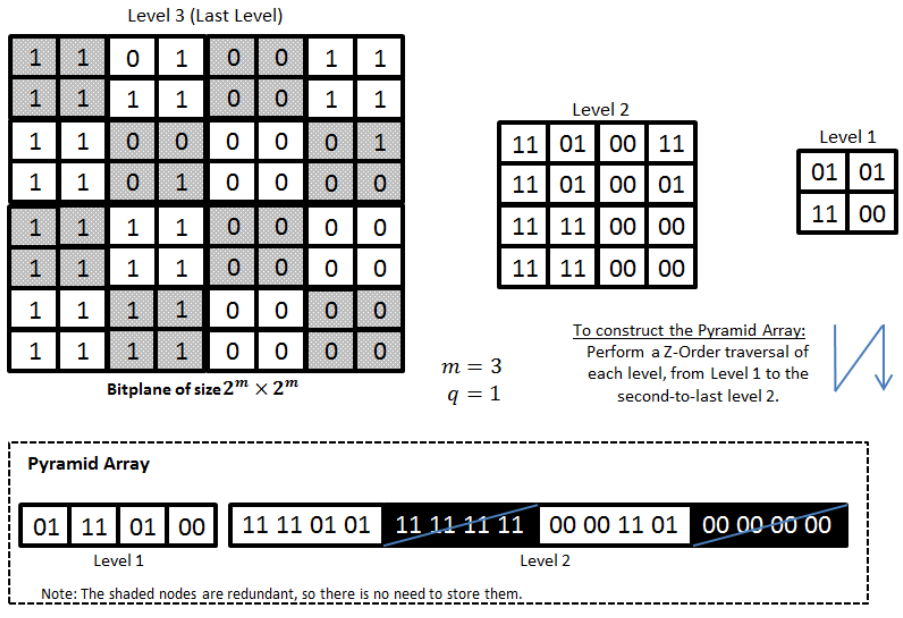

Figure 13: Construction of the Pyramid Array

3.1 Construction of the Pyramid Array

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, the codes for its four nodes are concatenated following the Z-order. As we can see in Figure 13, if a Z-order traversal of level 1 is performed, we would visit the entries of this level in the following order: (0,0), (1,0), (0,1) and (0,0). As this is done, the values of those entries are written to the pyramid array. So this is why in Figure 13, 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 is performed, which produces the entries 11, 11, 01, 01, 11, 11, 11, 11, 00, 00, 11, 01, 00, 00, 00, 00, which are concatenated to the entries of level 1. The same procedure is repeated for the following levels: for level *i*, we concatenate the codes for its $2^i \times 2^i$ nodes in Z-order, and append them to the linearization of level i-1 (see Figure 13 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]), the codes corresponding to the 2i×2i nodes in any given level *i* are grouped together into groups of 4 codes of 2 bits each. For example, in Figure 13 inside the pyramid array, a group of 4 entries at a time is created. The first group consists of 00, 11, 01 and 00, the second group consists of 11, 11, 01, 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 13 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 13 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.

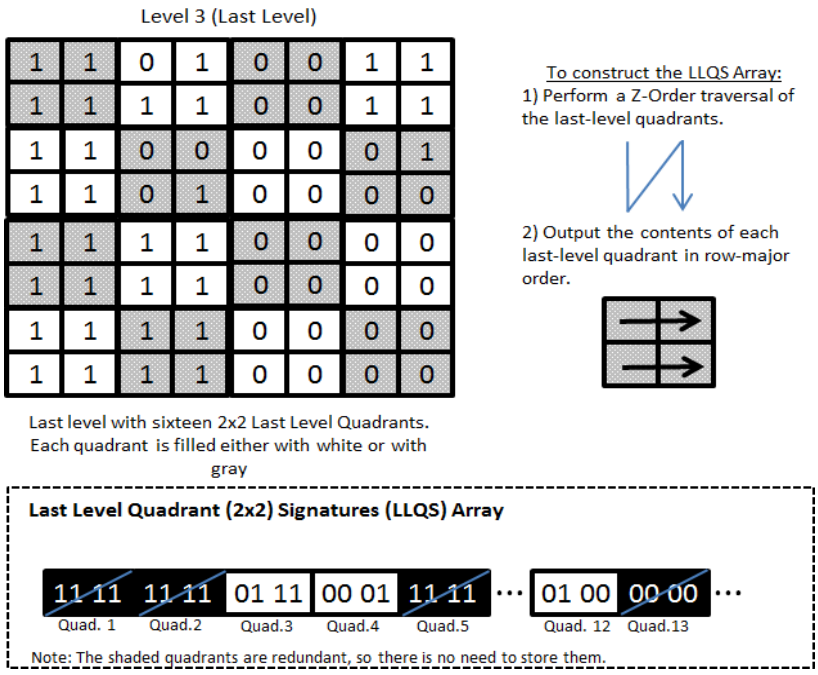

Figure 14: Construction of the LLQS Array

3.2 Construction of the LLQS Array

To construct the LLQS array, all quadrants are visited 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 a quadrant is entered, an output of its $2^q \times 2^q$ elements in row-major order is added to the LLQS array. This procedure is illustrated in Figure 14. In this figure, each last-level quadrant has size 2×2 and is visited in Z-order. So in Figure 14, the first quadrant that is visited is the one shaded gray in the upper left corner containing only 1s. Then its four 1s are written 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’. The same procedure is followed for the remaining 14 last-level quadrants.

Again, there is an opportunity to prune the information from the *LLQS* array. Every time a uniform last-level quadrant (having either all entries with 0s or all with 1s) is encountered, it can be discarded. The reason these entries can be pruned 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 9 where there are shaded entries that have been stricken out.

3.3 BQ-Tree compression is lossless

In this subsection we prove that BQ-Tree compression 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 the BQ-Tree construction can be reversed as was reported in [68]. 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 11 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 11111111.

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≥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 15 marked with a circle). Since this is the case, we know that at position 4(i+1) (shown in Figure 15, 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.

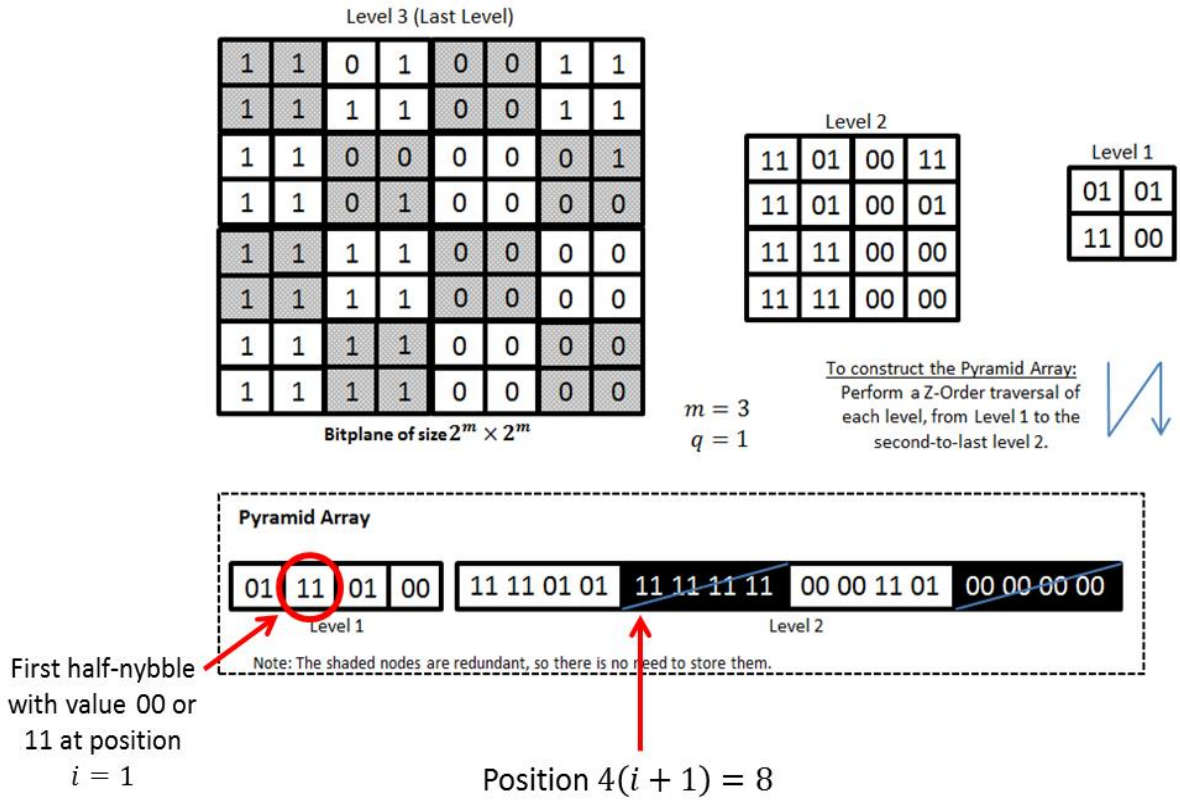

Figure 15: Illustration of proof of BQ-Tree Losslessness

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≥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. BQ-Tree Compression Issues

The BQ-Tree although cache-conscious through the use of a linearized array, still has a high time complexity. Indeed, for a raster of size N, on a single-core CPU, it will require a time complexity of *O(N)* to examine each value in the raster to construct the last level quadrant array (LLQS). Using multi-core CPU with *c* number of cores reduces the complexity to $(2^m x 2^m)/c$. However, with modern architectures, the number of cores is limited to only up to 32 cores, which means that for very large data the time complexity will be very high, therefore the time complexity remains *O(N)*. Fortunately, GPGPUs technologies provide an alternate way of parallelizing data processing by launching thousands of smaller and simpler dedicated cores in parallel. In the next chapter, we present parallel algorithms for the BQ-Tree on GPGPUs which aim to reduce the time complexity.

5. Conclusion

In this chapter we introduced the data structure used in our parallel algorithms. We point out issues in encountered with the sequential BQ-Tree algorithm. In the next chapter, we provide solutions to the compression time issue through the use of parallel algorithms; we provide detailed descriptions of the proposed algorithms.

Chapter 4: Parallel BQ-Tree Algorithms on GPGPUs

1. Introduction

In the previous chapter, we introduced the data structure that serves as a basis for our compression algorithms. In this chapter we give detailed descriptions of the two proposed algorithms: One Block per Tile (OBPT) and Multi-Block per Tile (MBPT). Before describing the parallel BQ-Tree algorithms, we present an overview of GPGPU technologies first, then we present important concepts used in the parallel algorithms and finally we explain the proposed algorithms.

2. GPGPU Overview

Graphic Processor Units (GPUs) are co-processors that carry out the necessary calculations to render graphical models. While performing this job, GPUs are required to execute the same piece of code, called shader, over millions of vertices under tight time-constraints [GK10]. In order to solve this problem, computer architects 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 [5].

The low cost of GPUs—both overall device cost and energetic cost per instruction (GPUs have higher performance per watt than CPUs[5])their high availability, and the fact that they have a throughput that is an order of magnitude greater than commercially available multicore chips [4] caught the attention of researchers from many different fields. Nonetheless, it was initially very difficult to harness the computational power of this hardware architecture because problems had to be casted in terms of a Computer

Graphics model. Noticing the difficulty faced by the research community in doing general-purpose computing on GPUs, GPU vendors, such as Nvidia, designed a C-like language CUDA, to facilitate the implementation of general-purpose algorithms on GPUs. GPUs that are CUDA-capable are part of a family of high performance processors conventionally called General Purpose Graphical Processing Units (GPGPU). In the subsections below, we will present an overview of the GPGPU programming model, GPGPU hardware implementation, the BQ-Tree implementation on GPGPUs and our proposed solutions to the GPGPU issues.

2.1 GPGPU Programming Model: CUDA

1.1.1 Kernels

The Computing Unified Device Architecture (CUDA) programming interface which supports the execution of general-purpose programs written in C, C++, FORTRAN, and other programs on NVDIA GPUs. Typically CUDA contains two types of code: code that runs on the CPU processor (host) and code that runs on the GPU processors (device) called kernels. Host CUDA code is compiled using standard compilers while the device code (kernels) is converted into a GPU intermediate language PTX, which later is translated into binary code that is optimized to run on GPUs [13].

A CUDA program typically starts with the execution of host code. Device code or kernels can only be launched from the CPU in this format: *kernel <<GridDim, BlockDim>>> (args)* where *kernel* is the name of the function to be launched, *GridDim* and *BlockDim* represents the total number and organization of threads, and *args* represents the arguments that are passed to the kernel function. *GridDim* provides the number of blocks to be launched and *BlockDim* gives the number of threads within a single block. Kernels

cannot run concurrently, meaning that all the threads assigned to a kernel have to finish executing and then a new thread allocation is performed for the next kernel to execute.

1.1.2. 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. A kernel which is defined in CUDA with this syntax: *kernel_name<<<dimGrid, dimBlock>>> (args)*.contains a set of instructions that each thread will execute step by step. 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. Blocks also have unique IDs that identify them at the grid level.

1.1.3 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.

As show in Figure 16, In a GPGPU there are 5 memory spaces that the programmer must take into consideration [5][13]: 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 100 cycles on average. The local memory, which is

allocated for each thread, takes 100s cycles on average to access. There is also the shared memory, which is accessed by all the threads in a block. Accessing shared memory takes 4-32 cycles on average. Finally, there is texture and constant memory 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, and only 4 cycles are required to fetch data; otherwise, it takes 100 cycles.

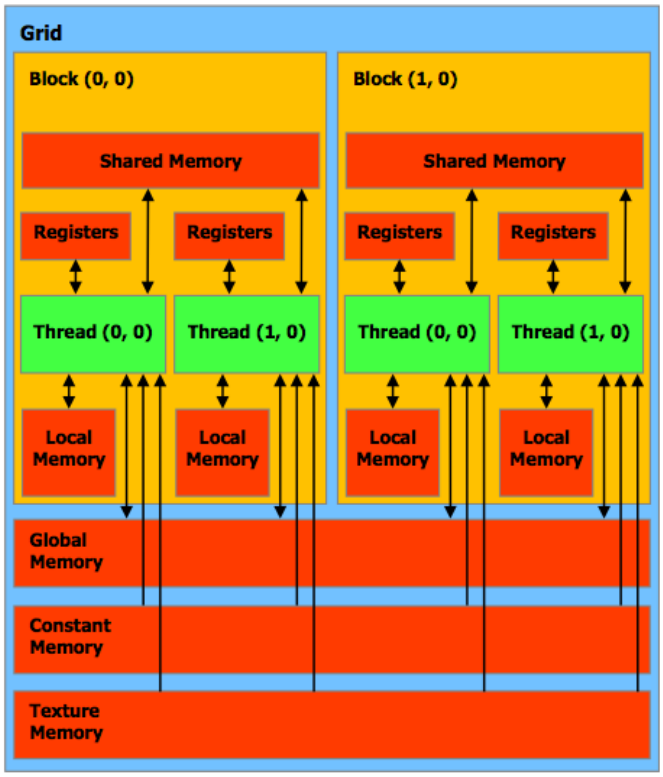

Figure 16: Logical overview of GPGPU resources

2.2 GPGPU Hardware Implementation

In our experiments are we used a SGI Octane III machine which is equipped with four Nvidia Fermi C2050. The Fermi architecture increases double precision operation

performance, and introduces true cache hierarchy as well as more shared memory. The Fermi hardware 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. The Fermi architecture supports up to 6GB of GDDR5 DRAM memory.

From the hardware specification given above and the threads hierarchy described in subsection 1.1.2, a hardware hierarchy is as follows: each GPGPU 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 blocks to allocate to each SM is decided by the GigaThread global scheduler based on the number of 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.

3. The Parallel BQ-Tree Algorithms

3.1 Introduction

Given that the compression of BQ-Trees may need to be performed multiple times during the execution of queries, and that the decompression of BQ-Trees is performed at most once, we chose to use a Last-Level Quadrant size of 4x4, which offers, for the compression of the NASA Modis North America raster that is used, a better compression ratio than the 2x2 LLQS size [36]. The characteristics of this dataset are explained in Chapter 5. We have two algorithms, the *multi-block-per-tile* and the *one-block-per-tile.* For both algorithms, the whole raster is first split into uniform tiles of size 1024x1024 or

4096x4096. For the *multi-block-per-tile* algorithm, 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 algorithm, 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 17 below, the BQ-Tree compression algorithm for GPGPUs is implemented in several kernels

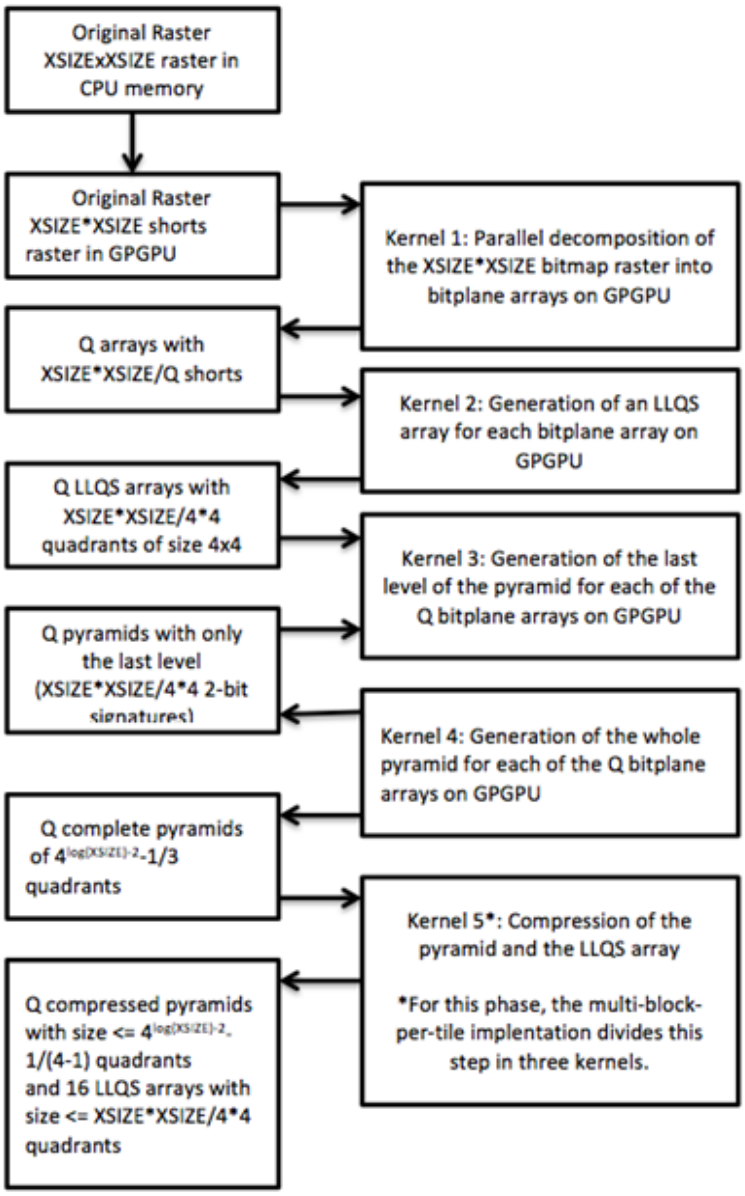

Figure 17: An illustration of the major steps of the generation of the BQ-Tree on GPGPUs for a raster of depth 16.

3.2 Definitions

A full pyramid: refers to a pyramid which has been fully constructed, with nodes in every level.

Tile: refers to a partition of the original bitmap of size $2^t x 2^t$. A bitmap partitioned into tiles will have $m/2^t * m/2^t$ tiles to represent it fully.

Grid: refers to GPGPU blocks organized in a one or two dimensional structure.

Block: refers to GPGPU threads organized in a one or two dimensional structure.

3.3 Important Concepts

3.3.1 Bit-wise Decomposition Scheme**:**

Given *x* values each with *b* bits, we want to combine together each i^{th} bit of each of the *x* cell values to create an array of *b* entries each with *x* bits. This is performed in the steps below using as an example an array *P* of 3 values each with 8 bits <1, 5, 7> which can be represented as <00000001, 00000101,00000111>:

1. An array *S* of *b* entries each with *x* bits is initialized, in our example, it would be an array S of 8 entries each with 3 bits.
2. For each value of *x* values in *P*:
    a. Get bit at position *b-1* and append it to *S [b-1]*. Using the values provided as example above, at the end of this step *S [b-1]* would contain *000*.
    b. The step above is repeated for each bit position b_i where *i = {2, b}.* At the end of these steps we will have 8 entries with *000, 000, 000, 000, 000, 011, 001, 111.*

This scheme is used during the decomposition phase of the bitmap and allows to divide a bitmap into multiple bitplane bitmaps.

3.3.2 Bit-wise Quadrant Construction Scheme**:**

Given *x* values each with *b* bits, the bit-wise quadrant construction scheme subdivides them into 2-dimensional regions of size $2^i * 2^i$ where *i* is any number between 0 and b bits contained in each value. The pre-condition to this process is that the number of input

values is equal to the number of bits contained within the values and the number of bits has to be a multiple of 2. The bit-wise quadrant construction scheme is performed in the steps below using an example to build an array of entries of size 4x4 using as input values <1,5,7,9,10,10, 15,14> each with 8 bits:

An array *S* of entries each with $2^i * 2^i$ bits is initialized where *i* is any number between 0 and $_b$ bits contained in each of the initial values. In our example, it would be an array S of 4 entries each containing 4x4 bits. To understand how this array S is laid out, we use Figure 18, the left-most figure shows how the array of entries is presented as input; however spatially, these values are adjacent in column-order. Therefore, along the y-axis we have a size of 8 bits and along the x-axis we have 8 bits.

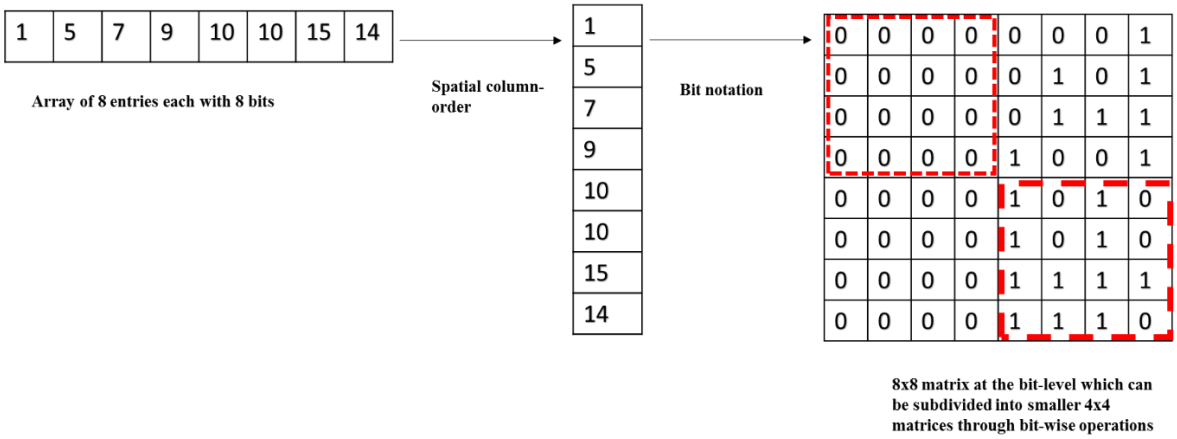

Figure 18: Bit-wise matrix of 8x8 size

To divide the 8x8 matrix in Figure 18, the following steps are performed.

1. The first 2^i entries are read; from our example, the first 4 (2^2) entries are 1, 5, 7, 9.
2. Read the first 4 (2^2) bits of each of the first four entries from the step above: 0000, 0000, 0000, 0000 (the region enclosed in a dashed line on the right-most

illustration shows these bits organized spatially). Append them one after the other in row-order and store them in S [0].

3. Read the second 4 (2^2) bits of each of the first four entries from the step above: 0001, 0101, 0111, and 1001. Append them one after the other and store them in S [3]. The reason they are stored in index 3 instead of index 1 is because we are using a z-order. How the z-order is calculated is by using the group of entries being processed as the row value and the group of bits being read as the y value. With the x and y values, we can generate the z-order. For our example, group 0 of entries is being read thus the row value is 0, and within this group, the second group of bits is being read, thus the column value is 1. The z-order of an area or point with row value of 0 and column value of 1 is 3 [1].
4. The step above is repeated for each group of 2^i entries until the whole array S is constructed.

3.3.3 Process Collectively and Loop:

For the construction of the pyramid described both in section 2.3.1.5 and section 2.3.2.5 below, the parallelization scheme that was chosen is Process Collectively and Loop (PCL) (described in more detail in [36]). This scheme is named this way because the group of working threads processes a set of elements by looping. For each iteration, 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 level *l*, all the threads execute a loop with *NumElementsInLevelL*/*NumThreads* steps. For example, in Figure 19, 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*.

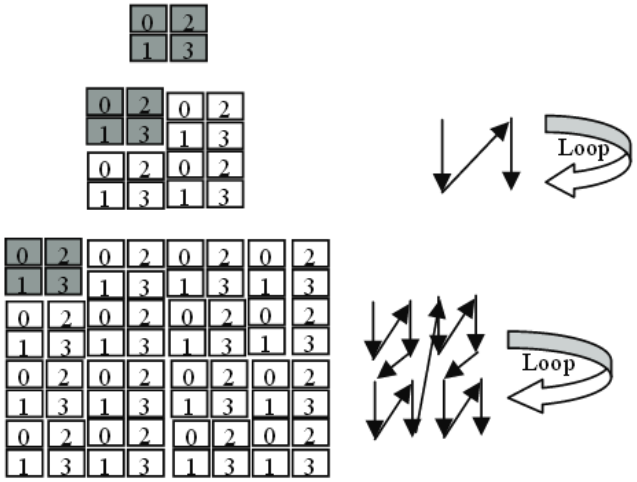

Figure 19: The Process Collectively and Loop Parallelization scheme

3.4 Algorithms

2.4.1. Multi-Block per Tile Algorithm (MBPT)

2.4.1.1 Structure Overview of MBPT

In this section, we present the components of 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 follow three major steps:

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

Secondly, on the GPGPU, the compression process starts with decomposing the tile, which is referred to as a bitmap from now on, into b distinct bitmap each with values containing one bit. Thirdly, for each of the bitplane bitmap, the LLQS array is constructed, at the end of this step, there are b LLQS arrays. Fourthly, 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 *b* BQ-Trees (composed of a compressed pyramid and an LLQS array). Finally, the *b* BQ-Trees are transferred to CPU memory.

2.4.1.2 Parallel Decomposition Step

The BQ-Tree compresses a bitmap at the bitplane level; therefore the first compression step is to partition the original bitmap into 16 bitplane bitmaps. This step takes as input a bitmap of depth 16 and provides as output an array of 16 bitplane bitmaps.

```
1. S[D]; O[D] // each thread creates an array S and O on its local memory with
   size D equal to the number of cell values that it will read at once
2. int col = threadIdx.x + blockIdx.x * blockDim.x; //each thread calculates its
   position within the grid using its block number and thread id
3. int row = threadIdx.y + blockIdx.y * blockDim.y;
4. for(int i = 0; i < D; i++)  //read input
            S[i] = bitmap [D * (row*bitmap_width + col) + i];
5. O = bit-wise decomposition (S) //decompose input
6. for(int i = 0; i < D; i++) // write output
            bitplanes[i * bitplane_size + row * bitplane_width + col] =
```

Figure 20: Pseudocode for MBPT parallel decomposition

Kernel:

Each thread executes the steps illustrated in Figure 20, it is important to note that these steps are sequential within each thread:

1. In Line 1, array S contains the input data to be decomposed and O will contain the output data after the thread processes S.
2. In Lines 2 to 3, each thread calculates its position within the grid, these global coordinates allow the threads to be mapped to the indices of the bitmap.
3. In Lines 4 to 5, each thread reads input into the S array, decompose each of the S array fields at the bit-level using bit-wise decomposition in Section 2.3.1.
4. On line 6, The last line sends each of the 16 16-bit values to its correct location in the bitplane bitmaps located in global memory array based on the indices calculated on Line 2 and 3.

2.4.1.3 Last Level Quadrant Array Construction

The Last Level Quadrant (LLQS) array contruction takes as input *b* bitplane bitmaps and outputs *b* bitplane LLQS arrays (Figure 21) and consists of the following steps:

5. In Line 1, array S contains the input data to be decomposed and O will contain the output data after the thread processes S.
1. In Line 2, using the block id the thread calculates which bitplane bitmap of the tile is supposed to work on. For example, threads in block 0 to 4 will process bitplane bitmap 0, block 5 to 16 will process bitplane bitmap 1 etc..
2. Line 3 calculates the offset to the correct bitplane, this allow all the threads in a block to access the bitplane assigned to it.
3. Lines 4 and 5 calculates the location of the thread at the grid level, this allows each thread to work on distinct values within the bitplane.
4. Line 6, using the offset to the bitplane and the block-level coordinates, read b consecutive values and store them in S.
5. On Line 7, using Bit-wise quadrant construction scheme described in section 2.3.2, we populate array O with the decomposed values from S.
6. On Line 8, Store array *O* to the correct bitplane LLQS array in global memory based on the bitplane offset and index calculated in Line 3 and 4.

```
1. S[D]; O[D];  // // each thread creates an array S and O on its local memory with
   size D equal to the number of values the thread reads
2. bitplane_index = get_bitplane_index(blockId); //get the bitplane that the thread
   is supposed to work on based on the thread block id.
3. bitplane_offset = bitplane_index * bitplane_size; // calculates the offset to the
   correct bitplane
4. const int row = threadIdx.y + blockIdx.y * blockDim.y;
5. const int col = threadIdx.x + blockIdx_x_mod * blockDim.x;
6. for(int j = 0; j < D; j++) {
       S[j] = bitplanes[ bitplane_offset + thread_loc (row, col) + j ];
7. O = bit-wise quadrant construction (S);
8. for(int i = 0; i < D; i++)
       llqs_d[bitplane _offset + z_order[thread_loc (row_col)] +i] = O[j];
```

Figure 21: Pseudocode for MBPT LLQS array construction

2.4.1.4 Last Level of Pyramid Construction

This step accepts as input *b* llqs arrays and outputs *b* pyramids with only the last level filled.

From Figure 22, the construction of the last level of a pyramid follows these steps:

1. From Line 1 to 2, based on the block id, generate the index of the bitplane the thread works on. Thread in block *i* will work on the *ith* bitplane. Calculate the offset to the correct bitplane LLQS as well as the correct bitplane pyramid to write to (on Line 3).
2. From Lines 4 to 7, calculate the offset to the correct location in the bitplane LLQs array to read from based on which iteration is currently being performed. Indeed the number of iterations is necessary because the number of threads might be lower than the size of the bitplane LLQS array being read.
2. From Lines 8 to 11, read the value at location *read_offset* and generate the signature based on the bit content of the value.
3. From Line 12 to Line 13, write the signature to a shared array *L* shared by all the threads in one block. Storing the values in shared memory allows to combine each

four signatures generated efficiently compared to performing this step on global memory.

4. On Line 14, once all the threads have read and written a signature to the shared array *L*, then each four 2-bit values in the shared array *L* are *XOR*ed together by one dedicated thread. This step allows to concatenate together siblings' nodes, each with 2-bits.
5. On Line 15, the contents of the shared array are then transferred to the Pyramid array in global memory in parallel. We offset to the last level by calculating the value contained in variable *beginning* using the geometric series sum in Equation 1 [69]. We subtract by 1 because the first node is considered to always be 1, thus it can be

omitted as being part of the tree and we divide by four because we write 4 signatures in one 16-bit value (each signature uses 2 bits of the 16 bits).

Equation 1: Offset to the last level of a pyramid

$$Beginning = (\sum 4^{l-1}) - 1/4 = \left[\left(\frac{4^{l}-1}{4-1}\right) - 1\right]/4$$

Where *l* is the current pyramid level being processed.

```
1. bitplane_index = blockIdx.x; // get the thread block id that the thread is part of
2. bitplane_llqs_offset = bitplane_index * bitplane_llqs_size; //calculate the offset
   to the correct bitplane llqs to read from
3. bitplane_pyramid_offset = bitplane_index * bitplane_pyramid_size; //calculate
   the offset to the correct bitplane pyramid array
4. iterations = bitplane_llqs/ blockDim.x; //calculate how many iterations to
   perform on this level of the pyramid
5. thread_index = blockDim.x * blockIdx.x + threadIdx.x; //calculate the index of
   the tread at the grid level
6. for each iteration in iterations: //for each iteration, read a value from the LLQS
   array and calculate a signature 0, 2, or 1 depending on the contents of the value
   read
7. read_offset =  (i * blockDim.x) + threadIdx.x;
8. p = llqs [bitplane_llqs_offset + read_offset;
9. if (p == 0X0000): signature = 0;
10. else if (p == 0XFFFF): signature = 2
11. else: signature = 1;
12. signature = signature << child_num_offset;
13. L[threadIdx.x] = signature;
14. parent_cell_value |= L[j + 4 * threadIdx.x]; *sequential by one thread*
15. pyramid_d[beginning + (i * blockDim.x/4) + threadIdx.x] =
parent_cell_value;
```

Figure 22: Pseudocode for MBPT construction of the last level of the pyramid

2.4.1.5 Pyramid Construction

This step accepts as input a pyramid array with only the last level filled and outputs a pyramid with all the levels filled. In section 2.3.2, we explained the Process Collectively and Loop scheme (PCL). The idea of PCL is to use all threads in a block to process the

same level of a pyramid. Threads will only build higher pyramid levels when the current level is completed.

```
1.  bitplane_index = blockIdx.x;
2.  bitplane_pyramid_offset = bitplane_index * bitplane_pyramid_size;
3.  for each level l of levels of pyramid:
4.  iterations = level_size/ blockDim;
5.  for each iteration in iterations:
6.  read_offset =  read_beginning+(i * blockDim) + threadIdx.x;
7.  p = pyramid [bitplane_pyramid_offset + read_offset];
8.  if (p == 0X00): signature = 0;
9.  else if (p == 0XFF): signature = 2
10. else: signature = 1;
11. signature = signature << child_num_offset;
12. L[threadIdx.x] = signature;
13. parent_cell_value |= L[j + 4 * threadIdx.x]; *sequentially by one block*
```

Figure 23: Pseudocode for MBPT construction of a full pyramid

The construction of the pyramid follows these steps as illustrated in Figure 23:

1. From Line 1 to 2, based on the block id, generate the index of the bitplane the thread works on. Thread in block *i* will work on the *ith* bitplane. Calculate the offset to the correct bitplane pyramid.
2. From Line 3 to 4, starting with the last level, each thread calculates the number of locations at level *l* that will be read at each iteration based on the size of a level which is calculated with this formula:

   $level\_size = 4^l$ With *l* being the current level being read from.
3. From Line 5 to 6, for each iteration, calculate the offset *read_beginning*, as shown in Equation 2, to the correct level *l* that will be read using the geometric series sum formula[69]:

Equation 2: Offset to level l of a pyramid

$$read\_beginning = (\sum 4^{l-1}) - 1/4 = \left[\left(\frac{4^{l-1}}{4-1}\right) - 1\right]/4$$

Where *l* is the current level being processed. This formula gives the cumulative number of nodes from the first level to the current level.

4. From Line 7 to 12, Read the value at location l and generate the signature based on the bit content of the value and store the signature a shared array L.
5. From Line 13 to Line 14, every four signatures are XORed together into one 8-bit value by one-fourth of the threads in a block. Then the results are written on the level above in the pyramid using geometric series sum formula:

Equation 3: Offset to lower level of pyramid

$$write\_beginning = (\sum 4^{l-2}) - 1/4 = \left[\left(\frac{4^{l-2}}{4-1}\right) - 1\right]/4.$$

This step is performed by one dedicated thread in each block.

6. Step 1 through 5 are repeated until all the levels are filled.

At the end of step 8, 16 full bitplane pyramids have been created in parallel.

2.4.1.6 Pyramid and LLQS Compression

Compressing the pyramid and LLQS array takes as input bitplane pyramids and LLQS arrays and outputs compressed bitplane pyramids and LLQS arrays. The compression of pyramids for MBPT requires synchronization among blocks. As such compression is performed in three main steps, these steps are the same for the compression of the LLQS array:

1. The first step takes as input all the bitplane full pyramids in one single large and linearized array and subdivides it into n sub-arrays. Each block is assigned a sub-array. Each thread reads a pyramid node within the sub-array, if a node contains a mixture of 1 and 0 bits (non-uniform pyramid node), then a “1” bit is recorded, otherwise a “0” is recorded in two separate arrays: B and T of the same size as the

pyramid. B and T are two arrays on global memory available to all the blocks in the grid. B is a Boolean array which stores “1” or “0” as a flag indicating a non-uniform or a uniform pyramid node, respectively. T is also a global array which records “1” or “0” in the same manner as B but these bits will later be used to calculate the location of nodes with signature 1 (non-uniform nodes). The number of non-uniform nodes in each sub-array is calculated by applying an inclusive scan [2] over each sub-array within *T*. Furthermore, the sum of non-uniform pyramid nodes in each sub-array T_i within *T* with signatures 1 are combined together in one array *R*. Indeed when an inclusive scan is applied on sub-array T_i, the last item in the array at position T_i*[numthreads-1]* contains the sum of all items in the sub-array T_i.

2. The second step takes as input array *R* and outputs an array *S* which is an exclusive scan of *R*; the last item *R[n-1]* contains the sum of all non-uniform nodes in the pyramid.
3. The last step takes as input array R and T and output a compressed array containing only pyramid nodes that have signatures 1. Using array *R*, the correct global offsets for non-uniform nodes are calculated. From step 1, *T* contains the locations of non-uniform nodes within their respective sub-arrays T_i. This step updates the nodes location from the sub-array scope to the whole array scope. Finally, all the non-uniform nodes are copied from the bitplane pyramids into a new array containing only non-uniform nodes.

2.4.2 The One-Block per Tile Algorithm

2.4.2.1 Structure Overview of OBPT

Before the compression steps are undertaken, all th tiles are transferred from CPU memory to GPGPU memory. If a raster contains x tiles, they are all transferred to GPGPU memory before any compression is started. The GPGPU start by assigning a block to each of the tiles in of the raster data.

Secondly, on the GPGPU, the compression process starts with decomposing *x* tiles in parallel, which is referred to as a bitmaps from now on, into *x.b* distinct bitplane bitmaps each with values containing one bit. Thirdly, for each of the *x.b* bitplane bitmaps, the LLQS array is constructed, at the end of this step, there are *x.b* LLQS arrays. Fourthly, each LLQS array is used to generate the last level of the pyramid and then the whole pyramid creating *x.b* pyramids. The last two steps compress the pyramid and the LLQS array for each bitplane bitmap creating *x.b* compressed pyramids and *x.b* LLQS. At the end of this algorithm, the output is *x.b* BQ-Trees (composed of a compressed pyramid and an LLQS array). Finally, the *x.b* BQ-Trees are transferred to CPU memory.

[1]2.4.2.2 Parallel Decomposition Step

The BQ-Tree compresses a bitmap at the bitplane level, therefore the first compression step is to partition the original bitmap into *D* bitplane bitmaps. This step takes as input a bitmap of depth 16 and provides as output an array of *D* bitplane bitmaps.

```
1. S[D]; O[D] //S contains the input values that the thread reads and O contains the
   decomposed values which are the output
2. tile_offset = tile_index*tile_width*tile_height //calculate the correct tile to be
   processed
3. row_offset_read = row*tile_width; //line 3 and 4 calculate the offset to the correct tile
4. col_offset_read = col*tile_height;
5. for(int i = 0; i < D; i++)        //read D values from global memory
       S[i] = bitmap [D * (row*bitmap_width + col) + i];
6. bit-wise decomposition (S) //decompose the values read at the bit-level
7. for(int i = 0; i < D; i++) //write the results back to global memory
       bitplanes[tile_offset + i * bitplane_size + row_offset_write + col_offset_write] =
       O [i];
```

Figure 24: Pseudocode for OBPT Parallel Decomposition

Kernel:

Each thread executes the steps below as show in Figure 24:

1. On Line 1, each thread creates an array S and O on its local memory of a size equal to the number of cell values that it will read at once. S will serves as the array holding the input values and O will contain the output generated by the thread.
2. Line 2 calculates the index of which tile to read from as well as the offset to that tile. For example, all the threads in thread block 1 will compress tile 1.
3. Line 3 to 4, the location to read from within a single tile is calculated, this ensures that each range of values has a dedicated thread in the thread block to process them.
4. On Lines 5, *D* values are read in the row-wise manner and its content stored in the thread's local array S.

5. On Line 6, each thread decompose each of the S array fields at the bit-level using bit-wise decomposition in Section 2.3.1 and the results are stored in array O.
6. On Line 7, the contents of array O are copied to global memory.

2.4.2.3 Last Level Quadrant Array Construction

```
1. S[D]; O[D]; //S is the input array and O is the output array
2. tile_index = get_bitplane_index(blockId); //get the block Id of the thread block the
   thread belongs to
3. tile_offset = tile_index * tile_size; //offset to the correct tile
   //for each bitplane in the tile, build a bitplane LLQS
4. for each bitplane in bitplanes: for each i in bitmap_width/blockDim: for each j in
   bitmap_height/blockDimx.y;
   //get the offset to the locations that the thread will
5. row = threadIdx.y + i * blockDim.y;  read from
6. col =  threadIdx.x +  j * blockDim.x;
7. for(int j = 0; j < D; j++): //read the input values
       S[j] = bitplanes[ tile_offset + bitplane_offset + thread_loc (row, col) + j ];
8. bit-wise quadrant construction (S); //construct the LLQS array for the values read
9. for(int i = 0; i < D; i++)  //write the output to global memory
       llqs_d[tile_offset + bitplane _offset +z_order[thread_loc (row_col)] ++i] = O[j];
```

Figure 25: Pseudocode for OBPT LLQS construction

This step accepts as input *b* bitplane bitmaps and outputs *b* LLQS.

Based on Figure 25:

1. On line 1, each thread creates an array S and O on its local memory with sizes equal to the number of cell values that it will read at once. Array S will contain the input values read by the thread and array O contains the output produced by each thread.
2. On line 2 and 3, the block id determines which tile this thread should work on and the offset to its position in the array of bitplane bitmaps. Because each block process one tile, then the block will have to process sequentially the whole tile.

3. On line 4 to 6, the tile is processed in parallel through three major iterations: iterate through the bitplane; within each bitplane, iterate *i* times along the y-axis; and finally within each i^{th} iteration, iterate j times along the x-axis. For each iteration the thread calculates the offset to the targeted bitplane, row number, and column number.
4. On line 7 to 8, the thread then reads 16 consecutive values along the y-axis and store them in S.
5. On line 9, using the Bit-wise quadrant construction scheme described in Section 2.3.2, we populate array O with entries of size 4x4 arranged in z-order.
6. Write the contents of O to the bitplane LLQs arrays on the global memory.

2.4.2.4 Last Level of Pyramid Construction

Based on Figure 26, the construction of the last level of a pyramid follows these steps:

1. From line 1 to line 4, generate the tile index that should be read from based on the blockId and the offset to that tile as well as the offset to the correct bitplane pyramid to write to.
2. From line 4 to line 11, calculate how many reads to be performed for each bitplane, then starting with the first bitplane, read a 16-bit value from the LLQS and create a signature of 00, 10, or 01 depending on the bit components of the value read. Store this signature in a shared array *L*.
3. From line 12 to line 13, based on the z-order at the quadrant level of the value read, the 2 bits are shifted to the left z-order times. This allows the next step to concatenate sibling nodes together and each sibling node is in the correct z-order position.

4. On line 14, once all the threads have read and written a signature to the shared array, then each four 2-bit values in the shared array are XORed together in parallel unlike the MBQT algorithm which uses only one thread to perform this step.

The contents of the shared array are then transferred to the last level of the Pyramid array in the global memory in parallel starting at the offset value *beginning* calculated with ,which gives the size of the pyramid up to the last level. Then, the contents of the shared array are transferred to the global memory by the first ¼ threads in of the block

```
1.  Tile_index = blockIdx.x; //get the thread block id that the thread belongs to
2.  Tile_offset = tile_index * bitplane_llqs_size*number_of_bitplanes; //calculate the offset
    to the correct location of the tile's bitplane LLQS arrays
3.  bitplane_pyramid_offset =  tile_offset + bitplane_index * bitplane_pyramid_size;
    //calculate the offset to the correct location of the tile's bitplane pyramids where the result
    will be written.
4.  iterations = bitplane_llqs/ blockDim.x; //calculate the number of iterations that will be
    performed to read all the values in each bitplane LLQS array.
5.  thread_index = blockDim.x * blockIdx.x + threadIdx.x;
6.  for each bitplane in bitplanes:  //for each bitplane, read the bitplane LLQS array and
    generate signature 0, 1, 2 for each LLQS array value.
7.  read_offset =  (i * num_threads) + threadIdx.x; //calculate the location to read from in the
    LLQS array
8.  p = llqs [bitplane_llqs_offset + read_offset;
9.  if (p == 0X0000): signature = 0;
10. else if (p == 0XFFFF): signature = 2
11. else: signature = 1;
12. signature = signature << child_num_offset;
13. L[threadIdx.x] = signature;     //write signature to shared memory of the thread block.
    //write the contents of the shared array L to global memory
14. parent_cell_value |= L[j + 4 * threadIdx.x]; *in parallel by ¼ of the threads in a block*
15. pyramid_d[tile_offset+bitplane_pyramid_offset+beginning + (i * blockDim.x/4) +
    threadIdx.x] = parent_cell_value;
```

Figure 26: Pseudocode for OBPT last level pyramid construction

2.4.2.5 Pyramid Construction

```
1.  tile_index = blockIdx.x;
2.  tile_offset = tile_index * bitplane_pyramid_size*number_of_bitplanes;
3.  for each bitplane in bitplanes:
4.  bitplane_pyramid_offset = bitplane_index * bitplane_pyramid_size;
5.  for each level l of levels of pyramid:
6.  iterations = level_size/ blockDim.x;
7.  for each iteration in iterations
8.  read_offset = read_beginning+(i * blockDim.x) + threadIdx.x;
9.  pyramid [bitplane_pyramid_offset + read_offset];
10. if (p == 0X0000): signature = 0;
11. else if (p == 0XFFFF): signature = 3
12. else: signature = 1;
13. signature = signature << child_num_offset;
14. L[threadIdx.x] = signature;
15. parent_cell_value |= L[j + 4 * threadIdx.x]; *in parallel by ¼ of threads in a block*
16. pyramid_d[write_beginning + (i * blockDim.x/4) + threadIdx.x] =
    parent_cell_value;
```

Figure 27: Pseudocode of OBPT construction of a full pyramid

This step accepts as input a pyramid array with only the last level filled and outputs a pyramid with all the levels filled. From Figure 27, these steps are performed:

1. On line 1 and 2, based on the block id, generate the tile index to read from and calculate the offset to the tile bitplane pyramids.
2. From line 3 to 4, for each i^{th} bitplane pyramid, calculate the offset to the ith bitplane pyramid.
3. From line 5 to 7, starting with the last level,

Calculate the number of nodes on the level using the equation below:

Equation 4: Size of level l

$$level\_size = 4^l$$

where *l* is the current level being processed,

and use it to calculate the number of iterations performed by all the threads in a block to process the whole level.

4. On line 8, calculate the offset *read_beginning* to the correct level *l* of the bitplane pyramid that will be read from where *l* is the pyramid level that is currently being read using Equation 1.
5. From line 9 to 14, based on the offset calculated in step 4, read a value, and generate a 2-bit signature based on the bit content of the value and store the signature in a shared array.
6. On line 15, in parallel, each four signatures and store them in the pyramid level above.
7. Step 1 through 6 are repeated until all the levels are filled.

Repeat the process above for each bitplane of the tile to create 16 filled pyramids.

At the end of the process above, at least 16*8 bitplane pyramids have been created. The number 8 represents the number of launched blocks in parallel on average for Fermi architecture. Therefore we can expect that at least 8 tiles have been processed to generate 16 full bitplane pyramids.

2.4.2.6 Pyramid and LLQS Compression

Because the pyramid is linearized just like the LLQS array, the steps are applied both to the pyramid and the LLQS array in a similar manner. Every time we use a pyramid, the same operation is applied to the LLQS array.

1. Take as input the bitplanes full pyramids and output the bitplanes compressed pyramids.
2. Based on the block id, generate the tile index to read from and calculate the offset to the tile bitplane pyramids.
3. For each *ith* bitplane pyramid, calculate the offset to the *ith* bitplane pyramid.

4. We assume that each bitplane pyramid will have at least the first four nodes in the pyramid (they only occupy an 8-bit character). We increment by 1 each position we read to make sure we do not compress the first 8-bits.
5. We then create three shared arrays, P, P_C and PTR. P and P_C. P contains the offset of each node of the pyramid, and P_C will contain only non-uniform nodes. PTR is used for holding the results of an exclusive scan which gives the total number of non-uniform nodes in the bitplane full pyramids.
6. In parallel, each thread reads a node from the bitplane pyramid and stores the value in shared array P.
7. Each thread reads a node from the shared array P and stores a 1 or 0 in the shared array PTR if the node is non-uniform or uniform, respectively.
8. An exclusive scan is performed on the shared array PTR to eliminate the offsets to uniform nodes, only remaining with non-uniform nodes offsets.
9. The last entry in PTR contains the total number of non-uniform nodes encountered in the sub-array.
10. Each thread reads again a single node contained in the shared array P at index *i*, if the node is non-uniform, then the offset contained in array PTR at index *i* is read and the node is written in sub-array P_C with offset PTR[*i*].
11. The last step copies the content of the shared array P_C to the global compressed pyramid location

Chapter 5: Performance Evaluation

Every chapter needs to have one or more open paragraphs explaining what this chapter is about and what the organization of the chapter is.

1. Experiment Environment

1.1 Hardware

As mentioned in the GPGPU overview in Section 2.2 in Chapter 4, in our experiments, we used a SGI Octane III machine which is equipped with four Nvidia Fermi C2050. The Fermi architecture increases double precision operation performance, and introduces true cache hierarchy as well as more shared memory. The Fermi hardware 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 1,536 threads. The GPGPU device connects with the CPU using a PCI-Express. The Fermi architecture supports up to 6GB of GDDR5 DRAM memory.

1.2 Software

In the performance analyses, we use datasets which have to first be transformed into the format we support (see section 2 below). This is achieved by using GDAL [41] which is an open-source C library that supports the translation of a wide range of geospatial data. We also use C++/NVIDIA CUDA to implement all of our algorithms. The code is compiled using the GNU Compiler [43].

1.3 Datasets

1.3.1 MODIS Satellite Images

The experiments were performed on the NASA MODIS (Moderate Resolution Imaging Spectroradiometer) raster datasets for Africa (referred to as B1 throughout the thesis) [37], North America (referred to as B2) [38], and Asia (referred to as B3 [39] that can be found at the Global Land Cover Facility (GLCF) website [40]. 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. We used these rasters because they are popular and big size for environmental science applications. We chose rasters with increasing sizes from B1 to B3 in order to observe the impact of the data size on our algorithms as well as the multi-core CPU algorithm. Figure 28 shows the NASA MODIS raster data that we used, the left-most map is shows a map of Africa and is referred to as B1, the second map in the middle shows a map of Northern Europe and is referred to as B2, and the right-most map shows Asia and is referred to as B3; their size increase in the order they are shown in the figure.

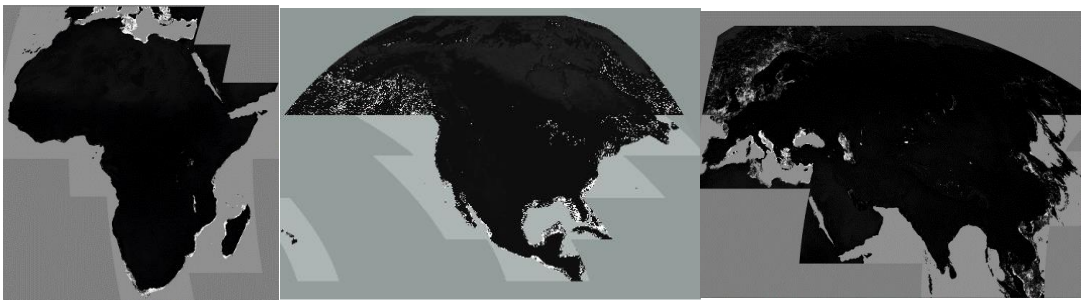

Figure 28: NASA MODIS raster datasets (B1, B2, and B3 in that order).

2. Experimental Model

2.1 Competing Technique

We compare our technique with another parallel GPGPU compression algorithm, the HFPaC described in the literature [chapter 2, section 9.3]. The HFPaC is a parallel compression algorithm which aims to compress height field data using Bezier curves and surfaces [60]. Bezier curves and surfaces are used extensively in computer graphics to create. This is done by first dividing the data into tiles of size 2^n+1*2^n+1, where n is a value in the range [7, 12] and each tile is processed independently. Secondly, each tile is

divided into segments of size $2^n+1 * 2^n+1$, where n is in the range [2, 5]. For each segment, Bezier 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 Bezier function (which uses the Bezier control points calculated) to generate an approximated value for the cell. Then the second and third layers contain the error between the Bezier approximated value and the actual 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 4: Feature Comparison of HFPaC and BQ-Tree

Feature	HFPaC	BQ-Tree
Lossy or Lossless	Lossless for select cases	Lossless
Type of Data	Height Field	Raster data
Byte Partitioning	b bits (after compression)	Bit-level (before compression)
Data partitioning	Tiles of size 2^n+1*2^n+1 where n is a value in the range [7, 12] and Rectangular segments of size 2^n+12^n+1where n is a size value in the range [2,5]	1024x1024 or 4096x4096 tiles

2.2 Performance Metrics

To measure the performance of the BQ-Tree, we evaluate a number of metrics.

Compression Time:

When we test our algorithm on GPGPUs, we calculate the total compression time for the whole raster data. We first compare both our proposed algorithms with each other. In this case we analyze the total time each version uses on the GPGPU. Second we compare our algorithms with the multi-core CPU one. For this analysis we measure the compression time for the parallel versions starting with the data transfer from CPU to GPGPU, the compression time on the GPGPUs, and the transfer time from the GPGPU to the CPU. This way we can have a fair comparison between the multi-core CPU algorithm and the parallel algorithms.

For comparing with the HFPaC, our competing algorithm, the compression time is measured starting with loading the data from the CPU to the GPGPU as well as the compression time. We do not include the GPGPU to CPU transfer time because the competing algorithm is implemented in a way that does not allow a fair representation of the data transfer from GPGPU to CPU.

Compression Ratio:

The compression ratio is calculated by dividing the original file size with the compressed format size. In the case of the BQ-Tree, we store not only the BQ-Tree but also a few metadata files used during decompression. The total compression ratio for the BQ-Tree is calculated as follows:

Compression Ratio: Original Raster Size/(BQ-Tree + Metadata)

The compression format of the HFPAC contains three layers (refer to Chapter 2, Section 8.2). The compression ratio is calculated as follows:

Compression Ratio: Original Raster Size/(Layer 1 + Layer 2 + Layer 3)

Average Query Processing Time:

The average query processing times were 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 iteration *i*, the raster is decompressed once, then *100*I* queries are ran one after the other and after they are completed, the total processing time is divided by the number of queries.

The total query processing time includes the average decompression time per iteration as well as the average query processing time. The decompression time does not include the time to load the compressed data from the CPU 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 CPU to GPGPU.

2.3 Query Definition

The query implemented is a spatial range query which is defined below:

Definition of Spatial Range, also referred to as window:

Given an *mxn* matrix $R=[r]_{ij}$ with real numbers, the window ${}^{m1,m2}_{n1,n2}WR$ 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 of Spatial Range Query (SRQ):

Given an *mxn* matrix $R=[r]_{ij}$ with real numbers and a window ${}^{m1,m2}_{n1,n2}WR$, the spatial range query *SRQ* $(R, {}^{m1,m2}_{n1,n2}WR)$ is defined as the set $\{(i,j), (r_{ij}) \mid (i,j) \in {}^{m1,m2}_{n1,n2}WR\}$ where r_{ij} is *ij* entry in matrix R.

The spatial range query is implemented in two main steps: a filtering step and a refining step.

The filtering step: Given a spatial range query with a window *W*, We first evaluate which tiles of the data are covered by the window W and these tiles. 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. From Figure 29, given a tile with 8x8 cells, the filtering step returns all the raster cells within the dashed line, which have a z-order within the z-order of the boundaries of the window indicated as the shaded cells.

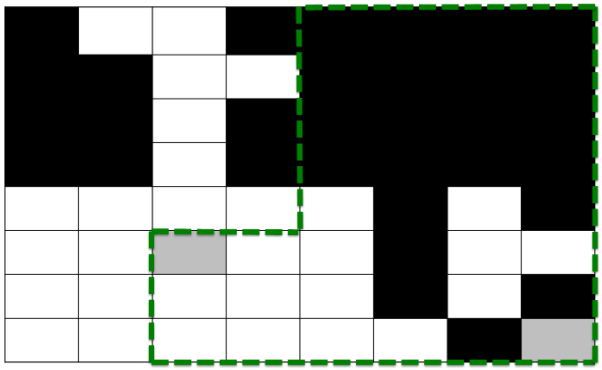

Figure 29: Spatial Range Query Filtering Step

Figure 30: Spatial Range Query Refining Step

The 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 it is performed on the CPU. Figure 30 shows the final cell values returned which are contained within the smaller shaded region.

3. Performance Results

3.1 Performance study of Compression Times of BQ-Tree on GPGPUs

3.1.1 Impact of resource allocation

GPGPU performance depends heavily on a number of factors, such as the resource allocation strategy, the amount of memory shared among resources, and the data transfer strategy between the CPU and the GPGPU.

In our performance analysis we focus on the resources allocation strategy as well as the transfer strategy. The reason is because the memory allocation strategy at the block level remains the same for both our proposed algorithms, therefore the change in performance is only affected by how many GPGUs resources we allocate.

For the multi-block per tile (MBPT) algorithm, the resource allocation strategy is to have all the blocks work together on one tile at a time. For the second strategy, one-block per tile (OBPT), each block is assigned to a single tile, therefore many blocks can process different tiles in parallel.

From Figure 31, we derive these results:

1. The OBPT BQ-Tree algorithm which allocates a number of blocks equal to the number of tiles available performs better for all tile sizes in terms of compression time.

2. The MBPT BQ-Tree algorithm which uses all the blocks on one tile at a time performs better on tile size 4096x4096 in terms of compression time, the reason being that for tile size 1024x1024, the allocated number of blocks is too low to take advantage of parallelization.

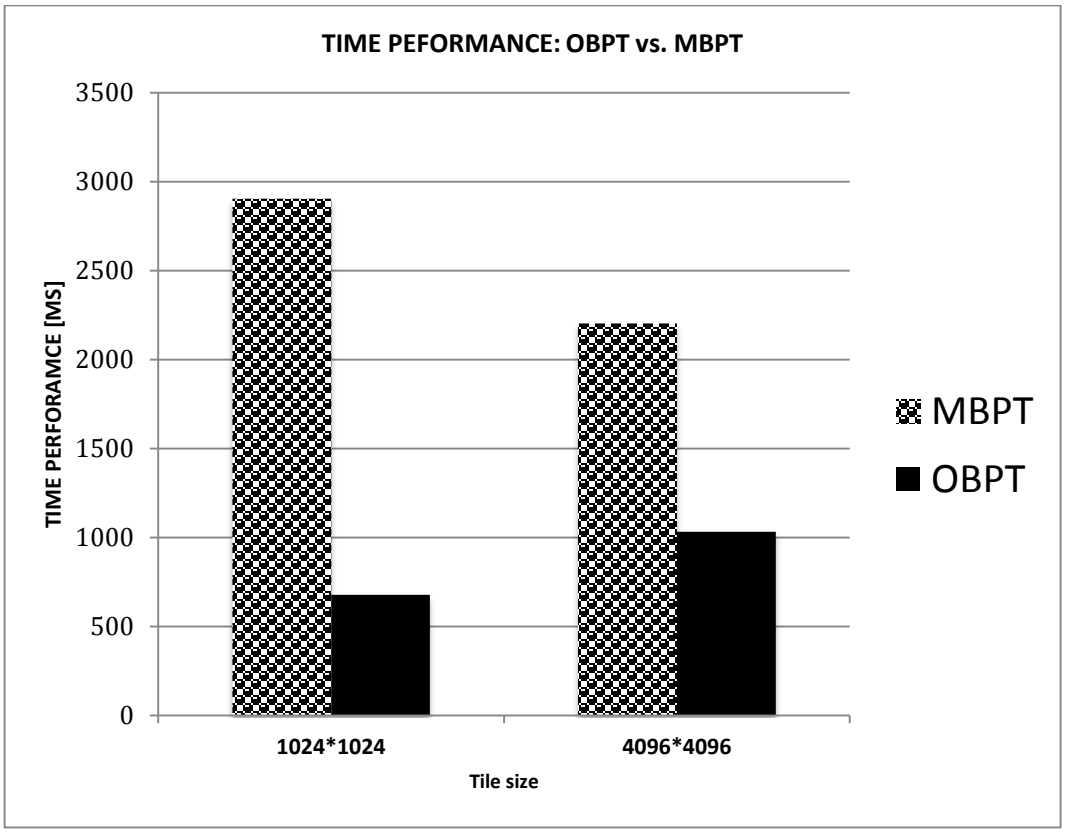

Figure 31: Overall compression time of the MBPT BQ-Tree and OBPT BQ-Tree

3.1.2 Impact of memory transfer strategy

For the MBPT technique, only one tile is loaded on the GPGPU memory at a time. For example, for the 1024x1024 tile size on a raster image of size 771 MB, there will be 368 tile transfers between the CPU 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.

Figure 32 shows the CPU to GPGPU transfer time for both the parallel algorithms of the BQ-Tree. The transfer times are calculated by running the BQ-Tree compression algorithm multiple times, using appropriate CUDA commands to gather the CPU to GPU transfer time, and then averaging the collected values. As stated in the paragraph above, the MBPT takes 368 loops to process the whole raster; and the OBPT transfers the whole raster image at once. For a tile size of 4096x4096, the MBPT takes only 24 loops to transfer the raster image and the OBPT sends the image in one take. Figure 32 shows that the CPU to GPGPU transfer time was reduced by 50% for MBPT. From Figure 32, the results show the following:

1. For the MBPT, the memory transfer improves with the size of the tiles. Bigger tile sizes implies fewer numbers of tiles, which reduces the number of transfers from the CPU to the GPGPU.

2. For the 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. Transferring the raster data from CPU to GPGPU in one step is similar to using up to 25 steps (representing the number of tiles of the largest dataset tested at a tile size of 4096x4096).

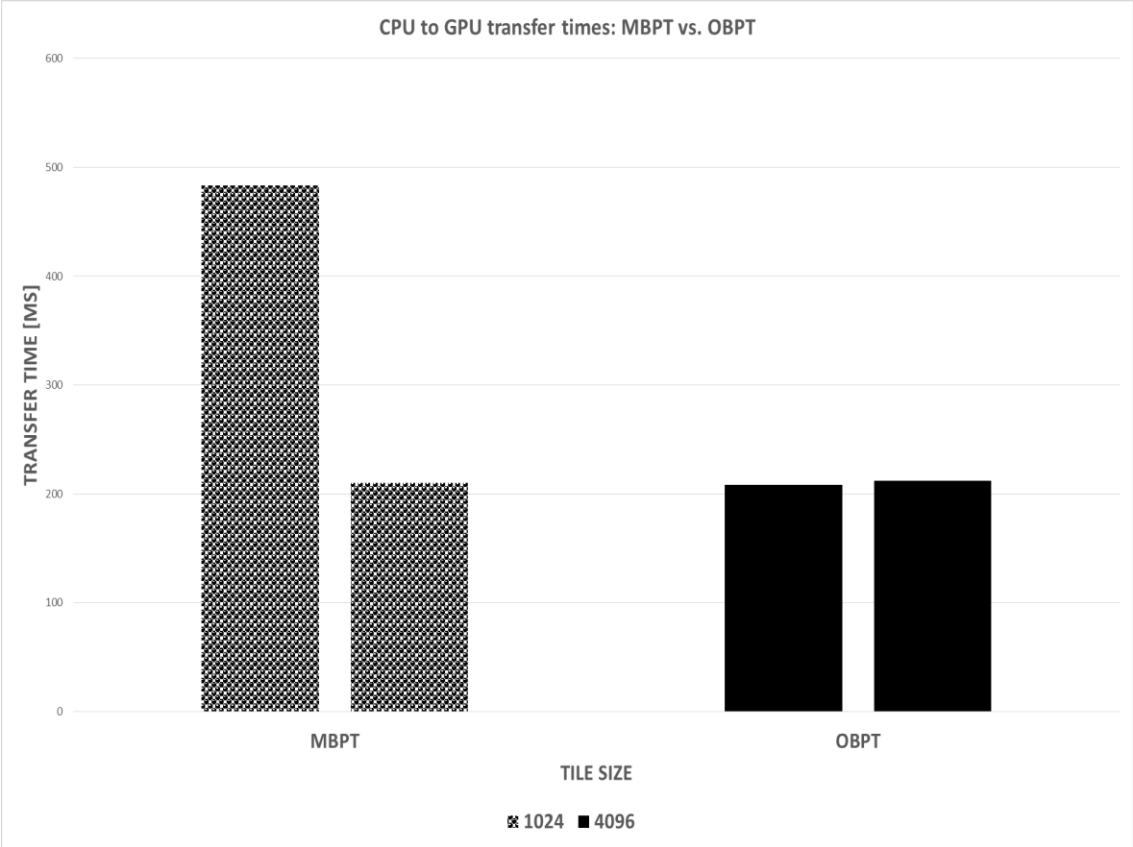

Figure 32: Comparison of the CPU to GPGPU average transfer time for MBPT and OBPT for 1024x1024 tile size and 4096x4096 tile size

3.1.3 Impact of dataset size

In a sequential algorithm, the execution time is expected to reduce as the dataset increases. However, for GPGPUs, execution time will only increase after all the resources have been allocated. To measure the scalability of parallel algorithms in relation to the problem size, we applied the multi-core BQ-Tree and the parallel BQ-Tree algorithms on datasets of varying sizes. The compression time is measured from end to end (compression time as well as transfer times between CPU and GPGPU) for the parallel BQ-Tree compression algorithms. For the multi-core CPU algorithm, we start timing after allocating the raster image array and stop the timing after the last tile has been processed.

From Figure 33 and Figure 34, we observe the following:

1. For a tile size of 1024x1024 the OBPT algorithm has the best compression times for all datasets sizes and its performance is linear across all tile sizes. The reason is that the resources allocation for both tile sizes is optimal for the OBPT algorithm. Indeed the blocks allocations (289, 368, 459 for B1, B2, B3, respectively) exceeds the maximum number of blocks that can be launched at once (which is 8 blocks for a Fermi architecture [2]) when using tiles of size 1024x1024.
2. The same behavior as above is observed for the OBPT algorithm for the tile size of 4096x4096 (25, 26, and 35 blocks are allocated for B1, B2, and B3, respectively). The MBPT compression time performance is linear across different dataset sizes, but performs much better with a tile size 4096x4096 (refer to Section 3.1.1 for a detailed explanation).
3. The multi-core BQ-Tree algorithm compression time performance is slightly exponential as the dataset size is increased. Additionally, we observe a deterioration in performance for tile size 4096x4096. The explanation lies in the way the multi-core BQ-Tree is implemented (refer to Section 2.1). For a tile size of 1024x1024, the work is divided among 16 cores which speeds up the sequential execution time. When a tile size of 4096x4096 is used, the multi-core implementation can only assign up to 16 threads, unlike the parallel GPGPU BQ-Tree algorithms which can launch up to thousands of threads.

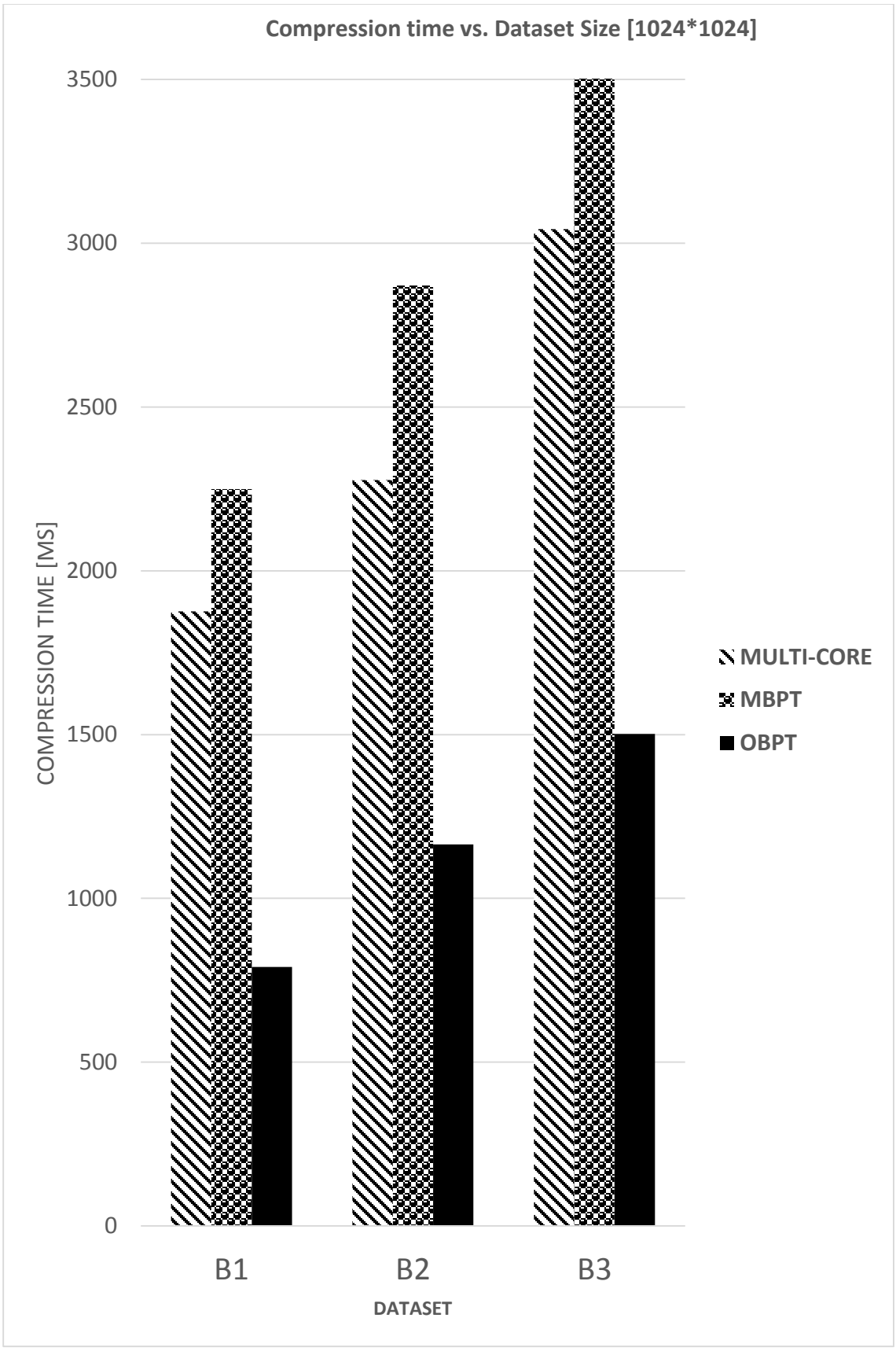
Compression time vs. Dataset Size [1024*1024]
3500
3000
2500
2000
1500
1000
500
0
COMPRESSION TIME [MS]
B1
B2
B3
DATASET
MULTI-CORE
MBPT
OBPT

Figure 33: Compression Time comparison between Multi-core BQ-Tree, MBPT BQ-Tree and OBPT BQ-Tree when varying dataset sizes

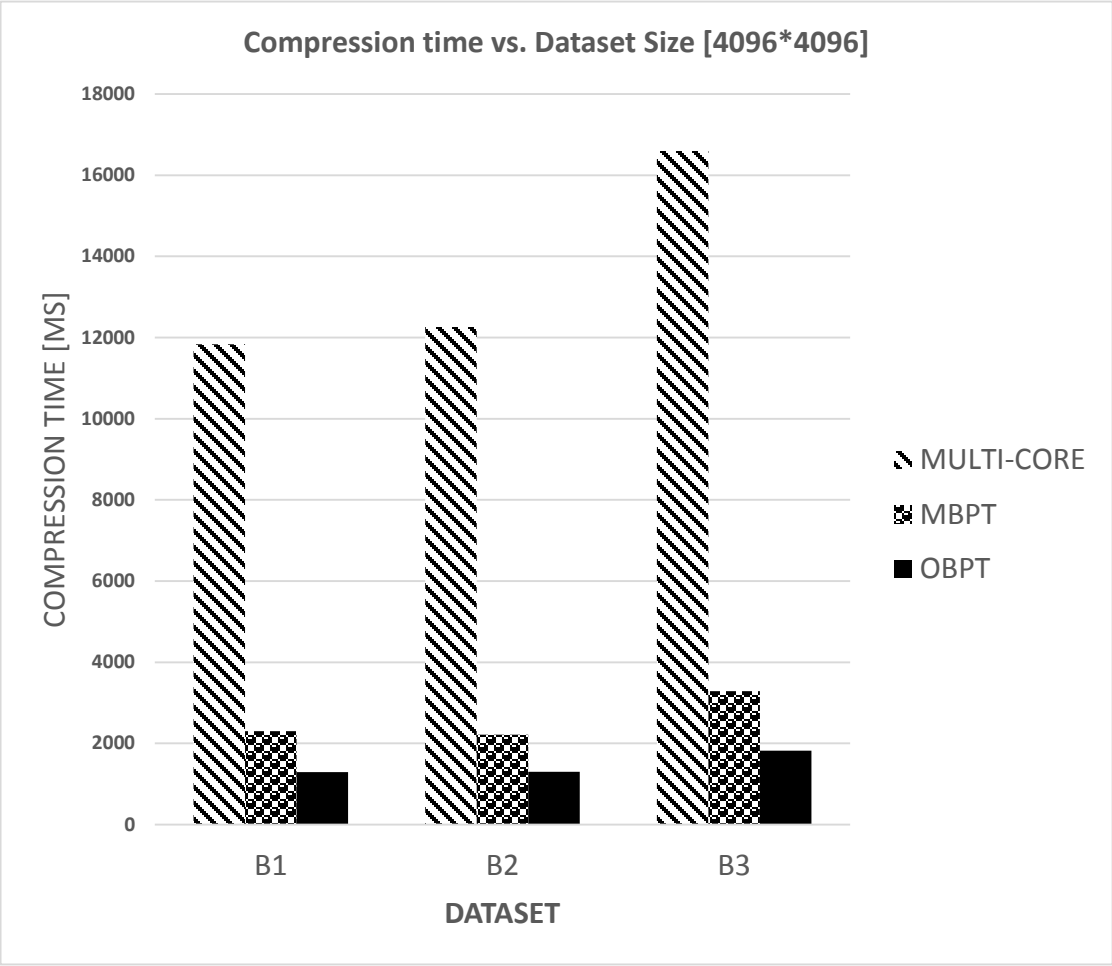

Figure 34: Compression Time comparison between multi-core BQ-Tree, MBPT and OBPT with varying dataset size

3.1.4 Impact of segment size

The HFPAC, our competitive compression technique, depends on different parameters such as the tile size, the segment size, and the level of error in the compressed format. In this study we compare only the lossless version of the HFPAC, therefore we do not need to use the level of error parameter. The HFPAC can handle data partitioned in tiles of sizes 2^n+1x2^n+1; where n is in range [7,12] we study the tile sizes $2^{10}+1*2^{10}+1$ and tiles

$2^{12}*2^{12}+1$ since they are most close to the tile sizes supported by the BQ-Tree and provide the most fair comparison with the HFPAC.

From Figure 35, we observe the following results:

1. HFPAC compression time increases as the size of segments used to calculate Bezier predicted values increases.

2. The Best HFPAC performance for a tile size of 1025x1025 is with segment size 5x5.

3, The OBPT outperforms the HFPaC across all segment sizes.

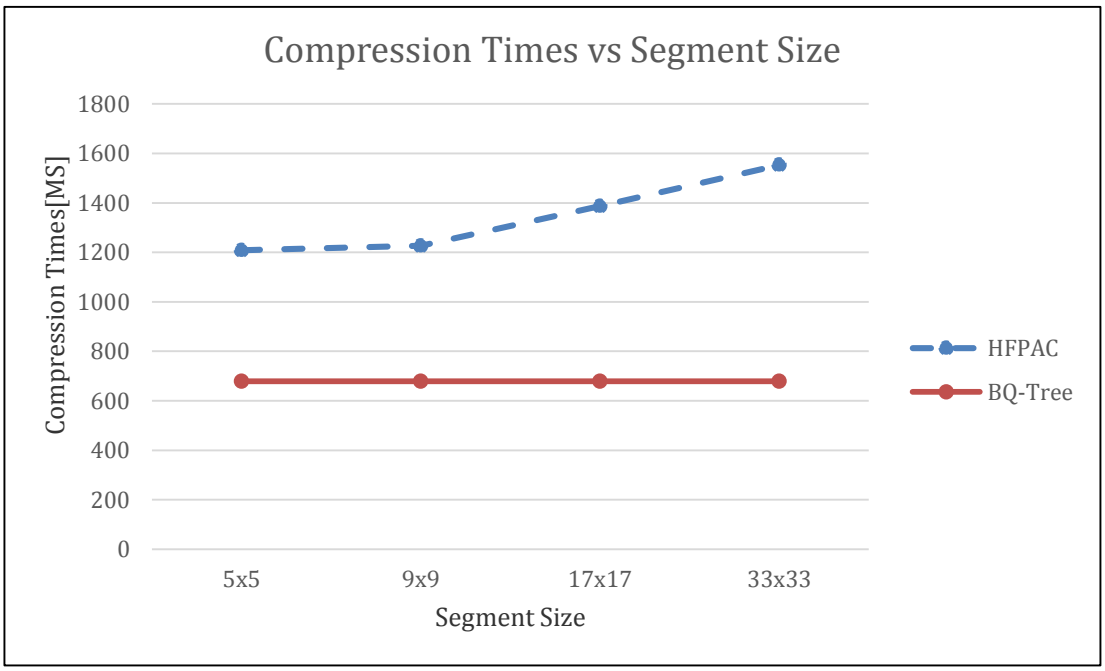

Figure 35: Compression Time comparison results of the HFPaC and the OBPT given varying segment sizes on tile size 1024x1024

Figure 36 show that:

1. On the data with tile sizes 4097x4097, the HFPaC performs faster on segment size 5x5.

2. As the segment size increases, the HFPaC compression time increases.

3. The OBPT outperforms the HFPaC across all segment sizes by 6X on average.

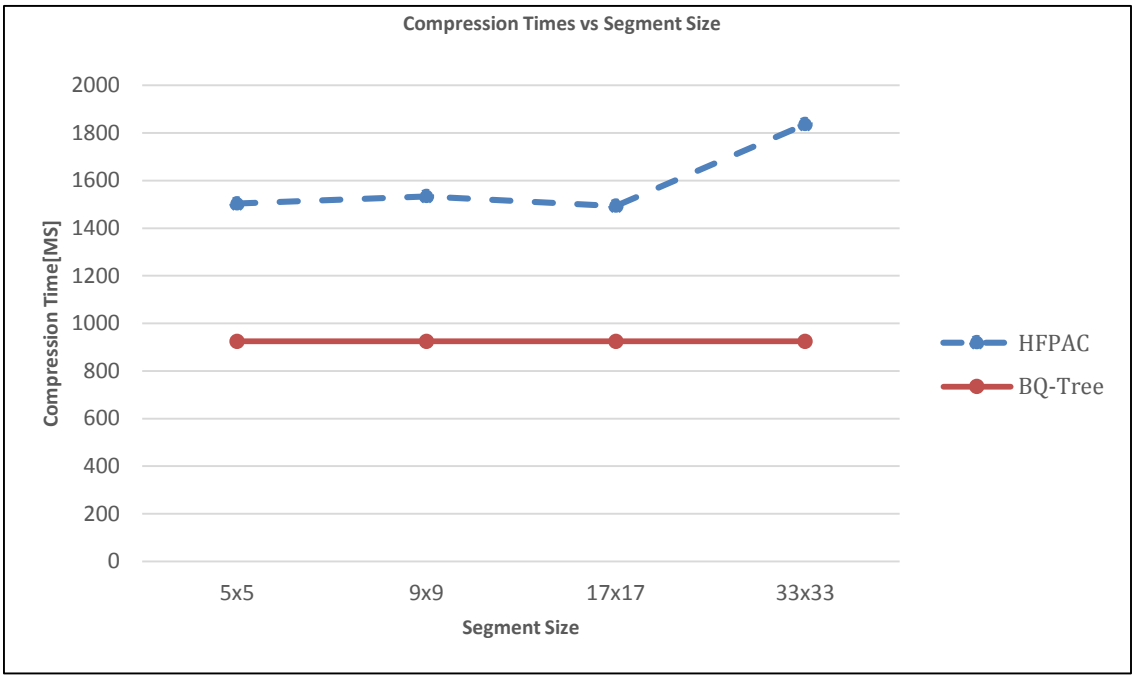

Figure 36: Performance results of the HFPAC and the OBPT given varying segment sizes and a tile size of 4096x4096

3.2 Performance Study of Compression Ratio of BQ-Trees on GPGPUs

We study the compression ratio of the OBPT by comparing it with our competitive algorithm, the HFPaC. For the 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. For the lossless version of the HFPaC, the compression ratio depends on more parameters: the size of the segments and the type of data.

We will measure the performance of the HFPaC on different segment sizes while keeping the OBPT performance the same.

From the performance results of compressing the MODIS dataset B2, shown in Figure 37, we observe the following results

1. The compression ratio of the OBPT is consistently better than the HFPaC compression ratio by a factor of 2 on average
2. The compression ratio of HFPaC is below 1, meaning that the compressed format is actually larger than the original data size.
3. The compression ratio of the OBPT is around 1:2, which reduces in half the initial data size.

3.2.1 Impact of Segment Size on Compression Ratio

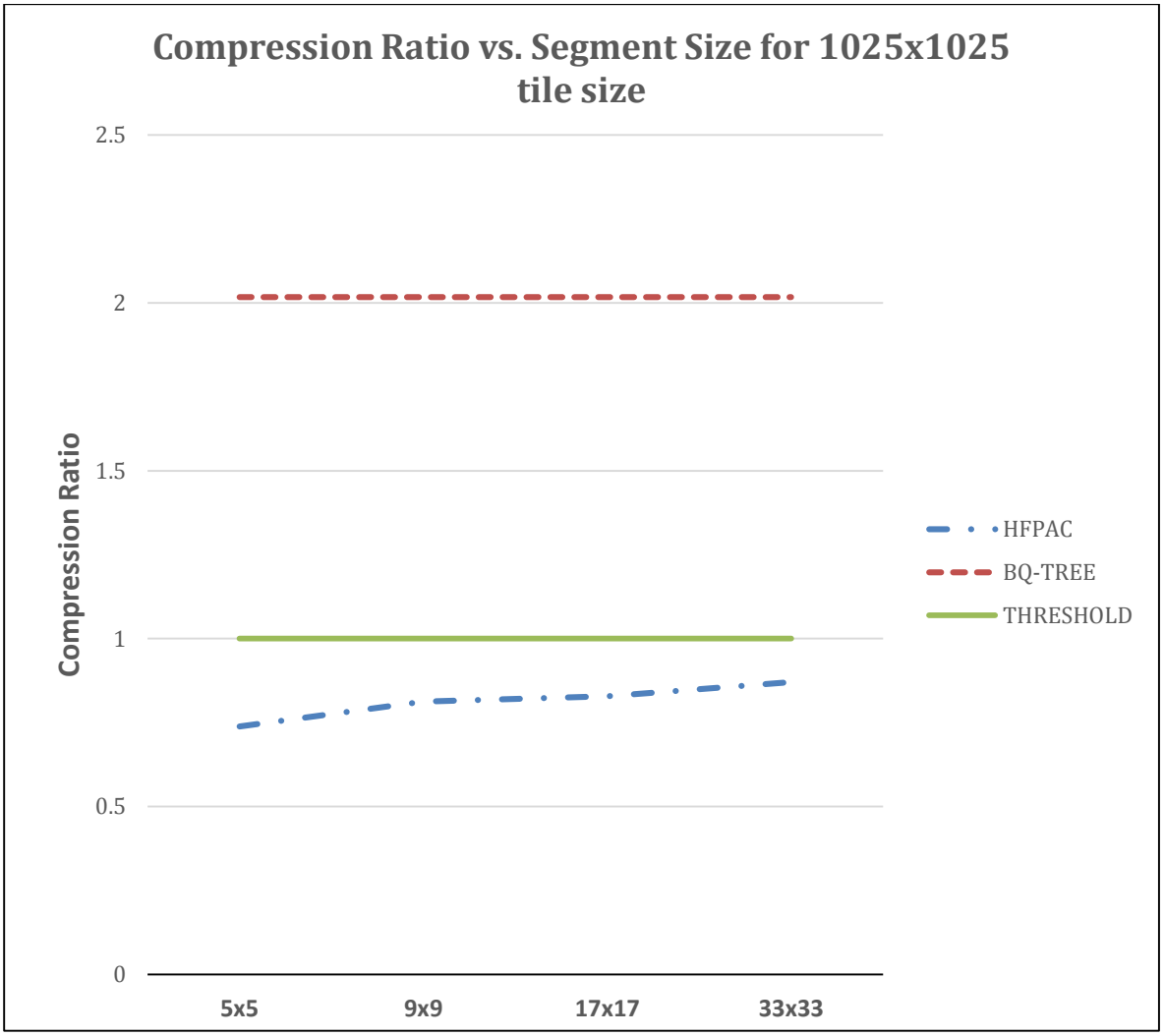

Figure 37: Compression ratio of OBPT compared to that of the HFPaC technique across different segments sizes on a 1025x1025 tile size.

3.3 Performance Study of the Average Query Processing 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 iteration *i*, the raster is decompressed once and then *100*I* queries are ran one after the other and after they are completed, the total processing time is divided by the number of queries. 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 B2 with tile size 1024x1024 and 1025x1025 for OBPT and HFPaC, respectively.

From Figure 38, we observe the following:

1. The impact of HFPaC compression on average query processing is higher than BQ-Tree compression by 1.23X on average.

2. As the number of queries is increased, the average query processing time is reduced. The reason is because that for each group of queries, the data only needs to be decompressed once. Therefore, as a group of queries grows in size, the decompression time remains the same but the time to run all the queries increases and the impact of decompression is reduced as a result.

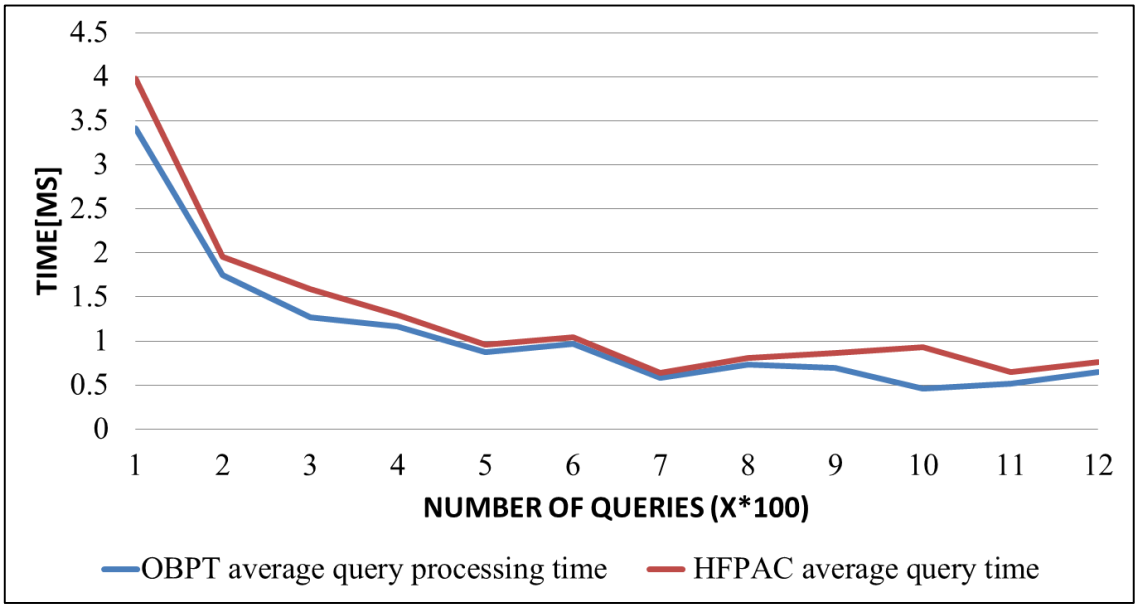

Figure 38: Average query processing times for OBPT and the HFPaC given an increasing number of queries

Chapter 6: Conclusions and Future Work

In this thesis we proposed two parallel compression algorithms on GPGPUs based on an improved version of the classic quadtree, the BQ-Tree. We provided a comprehensive study of popular compression techniques and explain the choice of the BQ-Tree. The proposed parallel algorithms of the BQ-Tree operate on partitioned data (tiled data). The first algorithm of the BQ-Tree, Multi-Block per Tile (MBPT,) allocates all the GPGPU resources on one tile at a time and the transfer of data between the CPU and the GPU is performed for each tile. The second parallel algorithm of the BQ-Tree, One-Block per Tile (OBPT), allocates resources on different tiles concurrently.

Both of these parallel versions are compared with the multi-core CPU BQ-Tree algorithms. The best algorithm is then compared with the state-of-the art parallel GPGPU compression algorithm, HFPaC.

1. Summary of Performance Results

Measuring the time performance of the BQ-Tree on GPGPUs requires the use of these parameters: tile size and dataset size. Furthermore, we compare the compression times for both parallel BQ-Tree algorithms with the multi-core CPU algorithm on datasets of different sizes. We also compare our best parallel BQ-Tree algorithm with a competing technique, the HFPaC. We measure both the compression time and the compression ratio of the best implementations of both algorithms.

From the performance analysis we can make the following conclusions:

1. The MBPT algorithm time performance improves by 1.31X when bigger tile sizes are used because the data transfer between the CPU and GPGPU is minimized

and the allocated resources are increased. Additionally, this behavior is expected because as the tile size increase the number of GPGPU resources allocated are increased.

2. The MBPT algorithm performs worse than the multi-core BQ-Tree algorithm on tile size 1024x1024, because the blocks and threads allocated are not enough to compete with the specialized 16 threads of the multi-core BQ-Tree algorithm. Indeed, the Multi-core CPU BQ-Tree compresses 1.2X faster on average compared to MBPT. These results were not expected because even if the amount of resources allocated is still low than the maximum possible allocation for GPGPUs, the number of threads launched in parallel is above 1024 threads on average, while the Multi-core CPU BQ-Tree algorithm uses only 16 cores. This indicate that the under-usage of GPGPU resources greatly impact the time performance of a parallel algorithm. This, coupled with a sub-optimal data transfer (transferring the data in small but multiple partitions) can make a GPGPU algorithm perform worse than a Multi-core CPU version of itself.
3. OBPT algorithm shows the best performance across all datasets and tile sizes because the use of the GPGPU resources is always maximized. It performs between 1.25X and 2.5X faster than MBPT, and between 3X and 9X faster on average compared to Multi-core CPU BQ-Tree.
4. OBPT performs better than the best implementations of the competing technique (HFPaC) for similar tile sizes on MODIS data. It performs on average 4X faster than the competing algorithm. This is expected because the HFPaC algorithm processes the height field by loading only one tile at a time on GPGPU memory.

Additionally, as the segment sizes increase the number of blocks allocated is reduced because the HFPaC resources allocation strategy allocates one block per segment. Therefre, as the size of segments increases, the number of segments decreases, which in turn affects the number of blocks allocated.

5. The BQ-Tree gives better compression ratio on MODIS datasets compared to the HFPaC which performs poorly since MODIS datasets do not exhibit a high mathematical smoothness [52]. These results are expected because the BQ-Tree datastructure can be applied on any data considered that it compresses at the bit-level. HFPaC, on the other hand, is optimized for height field data and as experiments showed in the previous chapter, it does not perform well in terms of compression ratio on non-height field data.
6. The impact of HPFaC compression on average query processing is higher than OBPT compression, meaning that the time to decompress the data for HFPaC is higher than OBPT. OBPT performs faster by 1.2X on average, therefore using the BQ-Tree for compression is more advantageous in terms of average query processing time.

2. Future Work

For future work, given that BQ-Tree compression both compresses and indexes data, 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 the CPU algorithm against our GPGPU algorithms of BQ-Trees, it would be particularly interesting to develop a hybrid CPU-GPGPU algorithm that makes use of the advantages offered by the two architectures. For example, we can use a CPU algorithm 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. Finally, we will also explore the scalability of this approach by using a multi-GPU setting or large GPGPU clusters.

References

1. Mamoulis N. Spatial Data Management. Morgan & Claypool, San Rafael, 2012.

2. Cuda C Programming Guide, 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

3. Hennessy, J.L. and Patterson, D. A. Computer Architecture: A Quantitative Approach. Morgan Kauffman, Boston, 2011.

4. Garland, M. and Kirk, D. B, Understanding throughput-oriented architectures. Communications of the ACM '10. 53(11): 58-66, 2010.

5. Lustig, D. and Martonosi, M., Reducing GPU offload latency via fine-grained CPU-GPU synchronization. HFPCA'13. IEEE Symp. On HPCA'13, 354-365, 2013.

6. Kirk, D. B and Hwiu, Wen-mei W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kauffman, San Fransisco, 2013.

7. Bakkum, P. and Skadron, K., Accelerating SQL database operations on a GPU with CUDA. in GPGPU'10. Proc. of the 3rd workshop on GPGPUs, 94-103, 2010.

8. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P, Relational joins on graphics processors. in ACM SIGMOD'08. Proc. of the int'l conference on Management of data, 511-524, 2008.

9. Gong, Z., Lakshminarasimhan, Jenkins, J., Kolla, H., Ethier, S., Chen, J., Ross, R., Klasky, S. and Samatova, N. F. Multi-level layout optimization for efficient spatio-temporal queries on ISABELA compressed data. in IPDPS'12, IEEE 26th symp. on IPDPS, 813-884, 2012.

10. Shook, E. and Wang, S. A parallel input-output system for resolving spatial data challenges: an agent-based model case-study. in ACM SIGSPATIAL'11, Proc. of 2nd int'l workshop n HPDGIS, 18-25. , 2011.

11. Achakeev, D. Seidemann, M., Schmidt, M. and Seeger, B, Sort-based parallel loading of R-trees, ACM SIGSPATIAL'12, Proc. of the 1st ACM SIGSPATIAL on Analytics for Big Geospatial Data, 62-70, 2012.

12. Luitjens, D. J and Rennich, D. S. CUDA warps and occupancy. in GPU Computing Webinar'11, 2011.

13. Kim, H., Vuduc, R., Baghosorkhi, S., Choi, J. and Hwu, W.-m. Performance Analysis and Tuning for GPGPUs. Morgan and Claypool, Boston, 2012.

14. Sayood, K. Introduction to Data Compression. Morgan Kaufman, Waltham, MA. 2012.

15. Solomon, D. Data Compression, The Complete Reference. Springer-Verlag, London, UK. 2007.

16. Storer, A. J. Image and Text Compression. Kluwer Academic, Boston, US. 1992

17. Magli, E. and Taubman, D. Image compression and standards for geospatial information systems, in IGARSS,'03, proc. of IEEE symposium on IGARSS, 2003.

18. Zhu, H. and C. P. Yang. Data Compression for Network GIS. Encyclopedia of GIS, 209-213.

19. Reid, M. M., R. J. Millar and N. D. Black. Second-generation image coding: an overview. ACM Computing Survey '97. **29**(1): 3-29.

20. McGarva, G., S. Morris and G. Janée (2009). Preserving Geospatial Data. DPC Technology Watch Series Report 09-01.

21. Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. Multi-Core Cache Hierarchies. Morgan & Claypool, .2011.

22. The GZIP homepage. http://www.gzip.org/

23. JPEG http://www.jpeg.org/

24. Lelewer, D. A. and Hirschberg, D. S., Data compression. ACM Comput. Surv., 19(3),261-296, 1987.

25. Rufai, A. M., Anbarjafari, G., Demirel, H., Lossy medical image compression using Huffman coding andsingular value decomposition. SIU'13. IEEE, Proc. of signal and communications applications, 1-4, 2013.

26. Dihong, T., Chen, W. H., Pi Sheng, C., Al Regib, G. and Mersereau, R.M., Hybrid variable length coding for image and video compression. in ICASSP'07, IEEE, I-1133-I-1136, 2007.

27. Sayood, K. Lossless Image Compression. Introduction to Data Compression (Fourth Edition). Morgan Kaufmann, Boston, US, 2012. P. 183-215, 2012.

28. Roth, M. A. and S. J. V. Horn. Database compression. SIGMOD Rec '93. **22**(3): 31-39. 1993.

29. Macedonas, A., D. Besiris, G. Economou and S. Fotopoulos. Dictionary based color image retrieval. Journal of Visual Communication and Image Representation. **19**(7): 464-470, 2008.

30. Rodrigues, N. M. M., E. A. B. da Silva, M. B. de Carvalho, S. M. M. de Faria and V. M. M. da Silva. On Dictionary Adaptation for Recurrent Pattern Image Coding. Image Processing, IEEE Transactions on **17**(9): 1640-1653, 2008.

31. Nelson, R. M. Data compression with the Burrows Wheeler Transform. Dr. Dobb's Journal '96. Vol. 21, No. 9. p.46, 1996.

32. Antonini, M.; Barlaud, M.; Mathieu, P.; Daubechies, I., Image coding using wavelet transform, Image Processing, IEEE Transactions on , vol.1, no.2, pp.205,220, Apr 1992

33. Cabeen, Ken, and Peter Gent. Image compression and the discrete cosine transform. College of the Redwoods, 1998.

34. Grossman, A. D. Information Retrieval: Algorithms and Heuristics. Springer, Norwell, MA. 2004.

35. Samet, H. Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann, San Fransisco, 2005.

36. Zhang, J., You, S., and Gruenwald, L. Parallel quad-tree coding of large-scale raster geospatial data on GPGPUs. in GIS'11, proc. of ACM SIGSPATIAL'11, 457-460, 2011.

37. NASA (2005), MODIS 16-day Composite MOD44C, Goodes. AF.2005225.tif. Collection 4, The Global Land Cover Facility, University of Marlyand, College Park, Maryland, Day 225,2005.

38. NASA(2005), MODIS 16-day Composite MOD44C, Goodes.EUAS.2003161.band1.tif,Collection 4, The Global Land Cover Facility, University of Marlyand, College Park, Maryland, Day 225,2005.

39. NASA (2005), MODIS 16-day Composite MOD44C, Goodes.EUAS.2003161.band1.tif, Collection 4, The Global Land Cover Facility, University of Marlyand, College Park, Maryland, Day 225, 2005.

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

41. GDAL- Geospatial Data Abstraction Library http://www.gdal.org/

42. GNU compiler. https://gcc.gnu.org/

43. Ebert, S.D. Texturing and Modeling: A Procedural Approach. AP Professional, Massachusetts, US, 1994.

44. USGS, The National Map. http://nationalmap.gov/small_scale/mld/satvw0i.html

45. Guttman, A., R-trees: a dynamic index structure for spatial searching. in SIGMOD '84. proc. of SIGMOD '84, 14(2), 47-57, 1984.

46. Ooi, B. C, Tan, K.-L., B-trees: bearing fruits of all kinds. Australian Computer Science Communications, 24(2), 13-20.

47. Comer, D. The Ubiquitous B-Tree. ACM Computing Surveys (CSUR) '79. 11(2);121-137, 1979.

48. Haoyu, T., Wuman, L., Huajian, M. and Ni, L.M. On packing very large R-trees. in MDM'12, proc. of 13th MDM, 99-104, 2012.

49. Goldstein, J., Ramakrishnan, R. and Shaft, U, Compressing relations and indexes. In ICDE' 98, proc. of 14th ICDE, 370-379, 1998.

50. 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 ICDE'12, proc. of 28th ICDE, 138-149, 2012.

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

52. Zhilin L., Zhu C., and Gold, C. Digital Terrain Modeling: Principles and Methodology. CRC Press, Boca Raton, FL. p. 75, 2010.

53. Storer, J. A., and Szymanski G.T. Data compression via textual substitution. Journal of the ACM (JACM) 29.4: 928-951, 1982.

54. Ozsoy, A.; Swany, M. CULZSS: LZSS Lossless Data Compression on CUDA. IEEE International Conference on Cluster Computing (CLUSTER). vol., no., pp.403,411, 26-30 Sept. 2011.

55. Anh N. V., Moffat A. Inverted Index Compression Using Word-Aligned Binary Codes. Information Retrieval, 8L 1510-166, 2005.

56. ZLIB. http://www.zlib.net/

57. Burtscher, M.; Ratanaworabhan, P. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. IEEE Transactions on Computers '09. vol.58, no.1, pp.18,31, Jan. 2009

58. O'Neil, M. A. and Burtscher, M. Floating-Point Data Compression at 75 GB/s on a GPU. Proc. Of GPGPU-4'11. Marc, 2011.

59. Simin, Y, Zhang, J. and Gruenwald, L. Parallel Spatial Query Processing on GPUs using R-Trees. Proc. Of BIGSPATIAL '13. November, 2013.

60. Andrzejewski, W. and 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: 315-329, 2010.Piegl, L.

61. Tiller, W. Curve and Surface Basics. The NURBS Book, Springer Berlin Heidelberg: 1-46, 1995.

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

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

64. Huffman Coding: An Example. http://www.binaryessence.com/dct/en000080.htm

65. Arithmetic Coding. commons.wikimedia.org/wiki/File:Arithmetic_coding_example.svg#file

66. The R-Tree, http://docs.oracle.com/html/A88805_01/sdo_intr.htm

67. Zhang, J. and You, S. A Quadtree-Based Lightweight Data Compression Approach for Processing Large-Scale Geospatial Rasters.

68. Kaligirwa, N. Leal, E, Gruenwald, L, Zhang, J., Simin, Y. Parallel Quadtree Encoding of Large-Scale Raster Geospatial Data on Multi-core CPUs and GPGPUs.

69. Cormen, H. T, Leiserson, E. C, Riverst, R. L., and Stein C. Introduction to Algorithms. The MIT Press, Cambridge, Massashusetts, 2009.