Published at http://www.tandfonline.com/doi/abs/10.1080/13658816.2013.828840

High-Performance Quadtree Constructions on Large-Scale Geospatial Rasters Using GPGPU Parallel Primitives

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

Abstract

The increasingly available Graphics Processing Units (GPU) hardware and the emerging General Purpose computing on GPU (GPGPU) technologies provide an attractive solution to high-performance geospatial computing. In this study, we have proposed a parallel primitive based approach to quadtree construction by transforming a multidimensional geospatial computing problem into chaining a set of generic parallel primitives that are designed for one dimensional arrays. The proposed approach is largely data independent and can be efficiently implemented on GPGPUs. Experiments on 4096*4096 and 16384*16384 raster tiles have shown that the implementation can complete the quadtree constructions in 13.33 milliseconds and 250.75 milliseconds, respectively, on average on an NVidia GPU device. Compared with an optimized serial CPU implementation based on the traditional recursive Depth-First Search (DFS) tree traversal schema that requires 1191.87 milliseconds on 4096*4096 raster tiles, a significant speedup of nearly 90X has been observed. The performance of the GPU based implementation also suggests that an indexing rate in the order of more than one billion raster cells per second can be achieved on commodity GPU devices.

1 Introduction

High-performance geospatial computing is an important component of geospatial cyberinfrastructure and is critical to large-scale geospatial data processing and problem solving (Wang and Liu 2009, Yang et al. 2010). Recently there is increasing interest in GPGPU technologies, i.e., General Computing on Graphics Processing Units, for high-performance geospatial data processing (Zhang 2010). High-end workstations equipped with GPGPU devices with hundreds and even thousands of processing cores that are capable of launching hundreds of thousands of threads simultaneously are ideal for massively data parallel, high-throughput and highly interactive applications in a personal computing environment. Recently Hong et al. (2011) argued that GPU architectures closely resemble supercomputers as both implement the primary Parallel Random Access Machine (PRAM[1]) characteristic of utilizing a very large number of threads with uniform memory latency (such as Cray XMT[2]). Solving small to medium sized problems directly on GPU-equipped personal workstations is both cost-effective and energy efficient. Equally important, as modern grid and cloud computing technologies increasingly rely on cluster computers made of identical computing nodes using commodity hardware, algorithms that can fully utilize GPGPU hardware capability on a single node will naturally boost the performance of cluster computers to solve larger scale problems.

Geospatial data processing on GPGPUs have attracted signficant research and application interest in the past few years ranging from data management to physics based environmental simulation. The throughput-oriented architectural designs of GPGPUs (Garland and Kirk 2010)

[1] http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine
[2] http://www.cray.com/products/XMT.aspx

are especially suitable for geospatial data processing due to the inherent parallelism of local and focal geospatial operations (Theobald 2005). However, it is generally nontrivial to use GPGPUs for zonal and global geospatial operations (Theobald 2005) whose parallelism can not be easily mapped to GPGPU computing architectures. Constructing tree indices to speed up query processing and data analysis is one of the most important operations in geospatial data processing which can be considered as a special type of global geospatial operation. Hundreds of tree indices have been proposed over the past few decades (Gaede and Gunther 1998, Samet 2005) and some of them have been efficiently implemented on CPUs.

The volume of raster geospatial data are increasing quickly. For example, the next generation geostationary weather satellite GOES-R serials[3] will improve the current generation weather satellite by 3, 4 and 5 times with respect to spectral, spatial and temporal resolutions (Schmit et al. 2009). With a temporal resolution of 5 minutes, GOES-R will generate 288 global coverages everyday for each of its 16 bands. At a spatial resolution of 2 km, each coverage and band combination has 360*60 cells in width and 180*60 cells in height, i.e., nearly a quarter of a billion cells. Such data volume growths are well above the computing power growth rate of uniprocessors. While Moore's law predicts that CPU computing power doubles every 18 months which has been true for more than 16 years before 2002, the growth rate of uniprocessors have dropped to about 20% per year from 2002 to 2006 and even lower in recent years (Hennessy and Patterson 2011). As such, it is natural to seek alternative parallel solutions to provide sufficient computing power to facilitate better understanding of the environments and their human impacts, including GPGPU technologies. Unfortunately, the current generation of GPGPUs have quite different hardware features than CPUs and it is nontrivial to port such algorithms from CPUs to GPUs. Despite the great potentials on using massively data parallel GPGPU technologies for geospatial computing, the performance of GPGPU-based parallel spatial indexing, including quadtrees for raster data, largely remains unknown to the geospatial computing community.

The work presented in this paper is a re-design and re-implementation of the Binned Min-Max Quadtree (BMMQ-Tree) construction algorithm for large-scale raster geospatial data that have been proposed previously (Zhang and You 2010a, Zhang et al. 2010). We show that by transforming a multi-dimensional geospatial computing problem into chaining a set of generic parallel primitives that are designed for one dimensional arrays (Section 2.2), we are able to reduce coding complexity and improve code efficiency at the same time. We believe that the new approach on parallel primitives based high-performance geospatial computing on GPGPUs can be interesting to geospatial computing researchers and developers who are seeking the parallel computing power of new hardware architectures but do not wish to be overwhelmed by hardware or programming model details. We hope the approach introduced in this paper can lower the barriers of applying GPGPU computing to efficiently solve practical geospatial problems and the example study reported in this paper can motivate similar research efforts. By generalizing the common patterns of applying generic parallel primitives in geospatial computing, more efficient geospatial-specific parallel primitives can be further developed. The rest of the paper is organized as the following. Section 2 introduces background and related work. Section 3 provides the design of the tree construction algorithm after reviewing the data layout of BMMQ-Trees. Section 4 presents the implementation details of BMMQ-Tree constructions on GPGPUs using parallel primitives. Section 5 reports experiment results and provides comparisons with alternative implementations. Finally Section 6 is the conclusion and future work directions.

[3] http://www.goes-r.gov/

Published at http://www.tandfonline.com/doi/abs/10.1080/13658816.2013.828840

2 Background and Related Works

2.1 GPGPU Computing and CUDA Programming Model

A Graphics Processing Unit (GPU) is a hardware device that is originally designed to work with a CPU to accelerate rendering of 3D or 2D graphics. The highly parallel structures of modern GPU devices, such as AMD/ATI Radeon[4] and Nvidia GeForce/Quadro series[5], make them more effective than general-purpose CPUs for a range of complex graphics-related algorithms. The concept of General Purpose computing on GPU (GPGPU) turns the massive floating-point computational power of a modern graphics accelerator's graphics-specific pipeline into general-purpose computing power. GPGPU computing technologies provide a cost effective alternative to cluster computing and have gained considerable interest in many research and application domains in the past few years (Hwu 2011a, Hwu 2011b). According to the Nvidia website, when compared with the latest quad-core CPU, Tesla 20-series GPU computing processors deliver equivalent performance at 1/20th of power consumption and 1/10th of cost[6]. As many reasonably current desktop computers have already been equipped with GPGPU enabled graphics cards, GPGPU based geospatial data processing can improve system performance significantly without additional costs. According to Garland and Kirk (2010), NVIDIA alone has shipped almost 220 million GPGPU-enabled devices from 2006 to 2010. Despite the differences among the GPGPU enabled devices and development platforms, a GPGPU device can be viewed as a parallel Single Instruction Multiple Data (SIMD)[7] machine. Major GPU hardware vendors have released Software Development Kits (SDKs) to facilitate application development using high-level programming languages. Among them, the Compute Unified Device Architecture (CUDA)[8] from Nvidia is arguably the most popular one which can be viewed as a C/C++ extension. The Accelerated Parallel Processing (APP) technology from AMD[9] is based on OpenCL[10] which is an open standard and is closely related to CUDA. We next briefly introduce the Nvidia GPU architecture and its parallel programming abstraction based on CUDA.

While different models of Nvidia GPU devices have different architectures, CUDA-enabled GPU devices are organized into a set of Stream Multiprocessors (SMs). Each SM has a certain number (e.g., 32) of computing cores. All the cores in a SM share a certain amount (e.g., 16k or 48k) of fast memory called shared memory and all the SMs have access to a large pool of global memory (e.g., 512MB or 4GB) on the device. According to CUDA, developers write special C-like code segments called kernels. The kernels are invoked by the companioning CPU code to run on GPU devices. CUDA based GPGPU programming makes it easier for task and data decomposition and subsequent parallel computing. Basically a developer specifies the sizes of the layout of the data to be processed in the units of data blocks and the number of threads to be launched inside a data block. The GPU hardware is responsible for mapping the data blocks to the SMs through space and time multiplexing which is transparent to developers/users. Since each SM has limited hardware resources, such as the number of registers, shared memory and thread scheduling slots, a SM can accommodate only a certain number of blocks subjected to the

[4] http://en.wikipedia.org/wiki/Radeon
[5] http://developer.nvidia.com/cuda-gpus
[6] http://www.nvidia.com/object/io_1227008280995.html
[7] http://en.wikipedia.org/wiki/SIMD
[8] http://www.nvidia.com/object/cuda_home_new.html
[9] http://developer.amd.com/sdks/AMDAPPSDK/
[10] http://www.khronos.org/opencl/

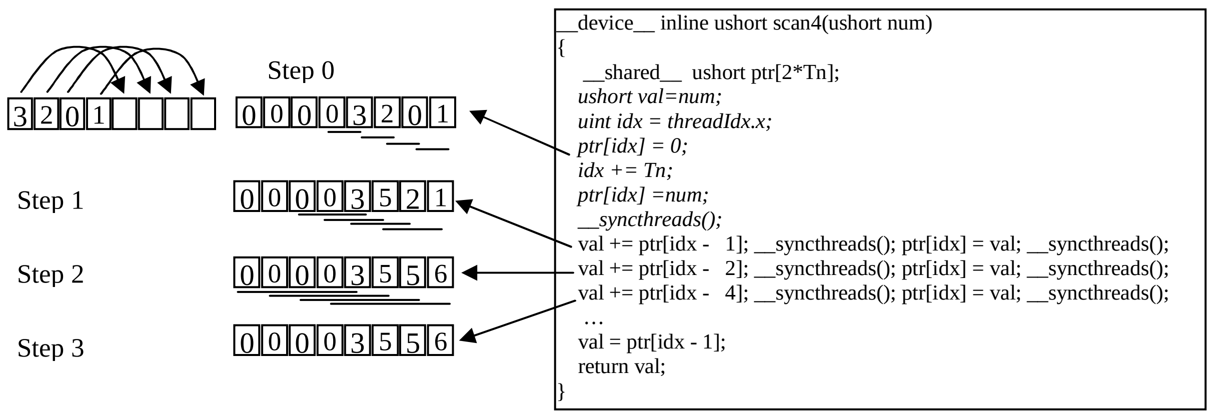
combination of the constraints. Carefully selecting block sizes allows a SM to accommodate more blocks simultaneously and, subsequently, improve parallel throughputs. While CUDA is designed to make parallel programming on Nvidia GPUs easier, due to the complexity of the massively data parallel hardware architectures, the learning curve of efficient CUDA programming can be steep. The Thrust library[11] that has been shipped with the latest CUDA SDK is designed to balance between easiness to use and code efficiency by providing a set of high-level APIs known as parallel primitives to be detailed next.

2.2 Parallel Primitives in the Thrust Library

Parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines. The behaviors of popular parallel primitives on one dimensional (1D) arrays or vectors are well-understood. We have opted to use 1D arrays in the context of geospatial computing to avoid confusing with vector geospatial data types. Unless explicitly stated, we use “arrays” to refer to “1D arrays”. Parallel primitives usually are implemented on top of native parallel programming languages (such as CUDA) but provide a set of simple yet powerful interfaces (or APIs) to end users. Technical details are hidden from end users and many parameters that are required by native programming languages are fine-tuned for typical applications in parallel libraries so that users do not need to specify such parameters explicitly. On the other hand, such APIs usually use template or generic based programming[12] techniques in a way similar to the well known Standard Template Library (STL)[13] so that the same set of APIs can be used for many data types. Due to the nature of high-level abstractions, the APIs may not be the most efficient ones when compared with handwritten programs using native programming languages with fine-tuned parameters. However, the APIs usually provide good tradeoffs between coding complexity and code efficiency. Indeed, most of the parallel primitives in the Thrust library are very similar to their STL counterparts and are very appealing to experienced STL users. The high level abstractions also bring signficant portability. In fact, while originally designed for CUDA-enabled GPUs, the latest Thrust library can also run on multi-core CPUs. This unique feature further makes parallel primitives based algorithm developments attractive when compared with using CUDA directly. While it is beyond the scope of this paper to provide a full introduction to parallel primitives and their implementations in the Thrust library (of which we refer to Bell and Hoberock 2011 and Thrust website), we next briefly introduce a few popular parallel primitives (McCool et al. 2012) that we will use in our quadtree construction design.

(1) *Scan*. The *Scan* primitive computes the cumulative sum of an array. Both the inclusive and exclusive scans are possible. For example, exclusive_scan([3,2,0,1])→([0,3,5,5]) while inclusive_scan ([3,2,0,1])→([3,5,5,6]). The *Scan* primitive can also take a user defined associative binary function to replace the default plus/sum binary function. To better illustrate the concept of the scan parallel primitive, a CUDA implementation of the scan primitive using four threads are provided in Fig. 1. In general, to scan 2^n data items, 2^{n+1} intermediate storage units are required. After the initialization step, the n data items are copied to the right half of the storage array while the first half of the storage array is cleared up. In step i of the process, data items that are 2^i elements away are added up in parallel and the whole scan process completes in n+1 steps.

[11] http://thrust.github.com/
[12] http://en.wikipedia.org/wiki/Generic_programming
[13] http://www.sgi.com/tech/stl/



```
__device__ inline ushort scan4(ushort num)
{
   __shared__  ushort ptr[2*Tn];
  ushort val=num;
  uint idx = threadIdx.x;
  ptr[idx] = 0;
  idx += Tn;
  ptr[idx] =num;
  __syncthreads();
  val += ptr[idx -   1]; __syncthreads(); ptr[idx] = val; __syncthreads();
  val += ptr[idx -   2]; __syncthreads(); ptr[idx] = val; __syncthreads();
  val += ptr[idx -   4]; __syncthreads(); ptr[idx] = val; __syncthreads();
   …
  val = ptr[idx - 1];
  return val;
}
```

Fig. 1 Illustration of *Scan* Implementation Using Four Threads in CUDA

(2) *Copy, copy_if, remove and remove_if*. *Copy* moves groups of elements from one location to another location, typically in two different arrays. The *Copy_if* primitive takes an additional unary function as a parameter to tell whether the corresponding array element should be copied to the output array or not. Similarly *remove* and *remove_if* remove groups of elements within an array with or without an optional binary predict function. *Remove* and *remove_if* are typically applied in-place which means that the input arrays can be the same as output arrays to save memory. Note that compacted arrays after applying *Remove* and *remove_if* primitives can be resized to reduce memory footprints.

(3) *Transform*. The basic form of *Transform* applies a unary function to each element of an input array and stores the result in the corresponding position in an output array. *Transform* is more general than *Copy* as it allows a user defined operation to be applied to array elements rather than simply copying. Thrust has also provide several variants of the *Transform* primitive to allow transform two arrays based on a binary function and/or using a separate stencil array to evaluate the criteria for transformation.

(4) Scatter. *Scatter* copies elements from a source range of an input array into an output array according to a map. For example, Scatter([3,0,2],[12,4,8],[*,*,*,*,*,*])→([4,*,8,*,12,*]). Note * values are those unchanged in the third array. Clearly when there is a one-to-one map between the inputs and outputs such as the Z-order transformation in our application, the output array will have no * values.

The alert readers many have observed that these parallel primitives work on flat 1D arrays only and we term them as generic primitives. From a geospatial computing perspective, this is indeed insufficient to process geospatial data which is usually multi-dimensional. However, as we shall show in Section 4, we can use these flat 1D arrays based generic parallel primitives as the building blocks to construct parallel geospatial processing modules. We note that the current generation of GPGPU devices work best with flat 1D arrays in many cases. The transformation between multi-dimensional geospatial data and flat 1D arrays can potentially help identifying parallelisms in geospatial computing and facilitate designing more efficient, geospatial-specific data structures and algorithms on GPGPUs for geospatial computing which is the key component in this paper.

2.3 Parallel Processing of Geospatial Data

Parallel processing of geospatial data is not a completely new concept. Quite a few works on parallel spatial data structures (Kamel and Faloutsos 1992, Ali et al. 2005), spatial join (Zhou et al. 1998, Patel and DeWitt 2000), spatial clustering (Xu 1999), spatial statistics (Armstrong et al. 1994, Wang and Armstrong 2003) and polygonization (Hoel 2003, Mineter 2003) have been reported. However, as discussed in (Clematis et al. 2003), research on parallel (and distributed) processing of geospatial data prior to 2003 has very little impact on mainstream geospatial data processing applications, possibly due to the accessibility of hardware and infrastructures in the past. The situation has been significantly changed over the past few years due to the wide availability of grid (Wang and Liu 2009) and cloud computing (Yang et al. 2011) resources and the maturity of GPGPU technologies (Zhang 2010). Work reported in (Wang et al. 2008) has demonstrated significant speedups by using grid computing for spatial statistics. Parallel computing on LIDAR data using cluster computers (Han et al. 2009) is getting increasingly popular due to its computation intensive nature. The development of a general-purpose parallel raster processing programming library on top of the MPI (Message Passing Interface[14]) parallel communication protocol is reported in (Guan 2010) and a test application using a geographical cellular automata model has achieved a speedup of 24 using a 32-node cluster computer. We also refer to (Yang et al. 2010) for a review on environmental modeling on cluster computers in a cyberinfrastructure environment. Recently, there are considerable research interest in geospatial data processing using the MapReduce parallel computing framework (Dean and Ghemawat 2010) and the open source Hadoop implementation[15] on cluster computers, such as R-Tree construction on point data and image tile quality computation (Cary et al. 2009), spatial join (Zhang et al. 2009b), geostatistics (Liu et al. 2010) and nearest neighbor queries on voronoi diagrams (Akdogan et al. 2010). Similar to MapReduce/Hadoop applications, there are also quite a few recent works on GPGPU applications to geospatial computing, including environmental modeling (Molna et al. 2010), flow accumulation (Ortega and Rueda 2010), drainage network computation (Qin and Zhan et al. 2012), LIDAR data reduction (Oryspayev 2012) and raster analysis (Steinbach and Hemmerling 2012). Most of these works are related to local or focal geospatial operations which are relatively straightforward to parallelize on GPGPUs. In addition, it seems that these works (except Molna et al. 2010) have focused on utilizing GPGPU's large number of threads to speed up computation but have not used GPGPU's high memory bandwidth and/or fast shared memory to speed up data accesses yet. As such, there are signficant potentials to improve the efficiencies of the respective implementation although it is nontrivial to understand data access patterns and make full use of GPGPU hardware capabilities. GPGPU technologies have also been applied for coding raster bitplane bitmaps (Zhang et al. 2011), polygon rasterization (Zhang 2011) and vector data indexing using R-Tree (Luo et al. 2011) where zonal and global geospatial operations are involved and more sophisticated parallelization schemes have been designed to optimize performance.

2.4 Indexing Raster Geospatial Data

There are relative fewer works on indexing raster geospatial data when compared with indexing vector geospatial data. Interval trees (Cignoni et al. 1997), octrees (Wilhelms and Vangelder 1992, Wang and Chiang 2009) and KD-trees (Gress and Klein 2004) have been extensively used in 3D computer graphics such as iso-surface rendering and ray-tracing.

[14] http://en.wikipedia.org/wiki/Message_Passing_Interface
[15] http://hadoop.apache.org/

Quadtrees have been proposed to compress binary and gray scale 2D rasters (Samet 1985, Lin 1997, Chan and Chang 2004, Chung et al. 2006). However, we note that data structures and algorithms designed for graphics rendering and image compression are not necessarily suitable for database query processing. Pyramid and tiling techniques have also been used to speed up image display but usually they do not allow queries on the underlying raster data. Oracle GeoRaster[16] allows storing the bounding boxes and derived attributes of tile images as vector geospatial data, which subsequently can be indexed and queried so that only selected tile images need to be retrieved for display. A few of existing works have addressed the issue of managing a set of similar/related rasters for efficient query processing based on the concept of overlapping quadtrees (Tzouramanis et al. 1998, Manolopoulos et al. 2001, Manouvrier et al. 2002). All the above indices construction algorithms are serial. It is desirable to investigate how modern GPU hardware devices and GPGPU parallel computing technologies can be effectively used to index large-scale raster geospatial data to support efficient queries. Unfortunately, as GPGPU are relatively new technologies to the spatial data management community, the performance is largely unknown.

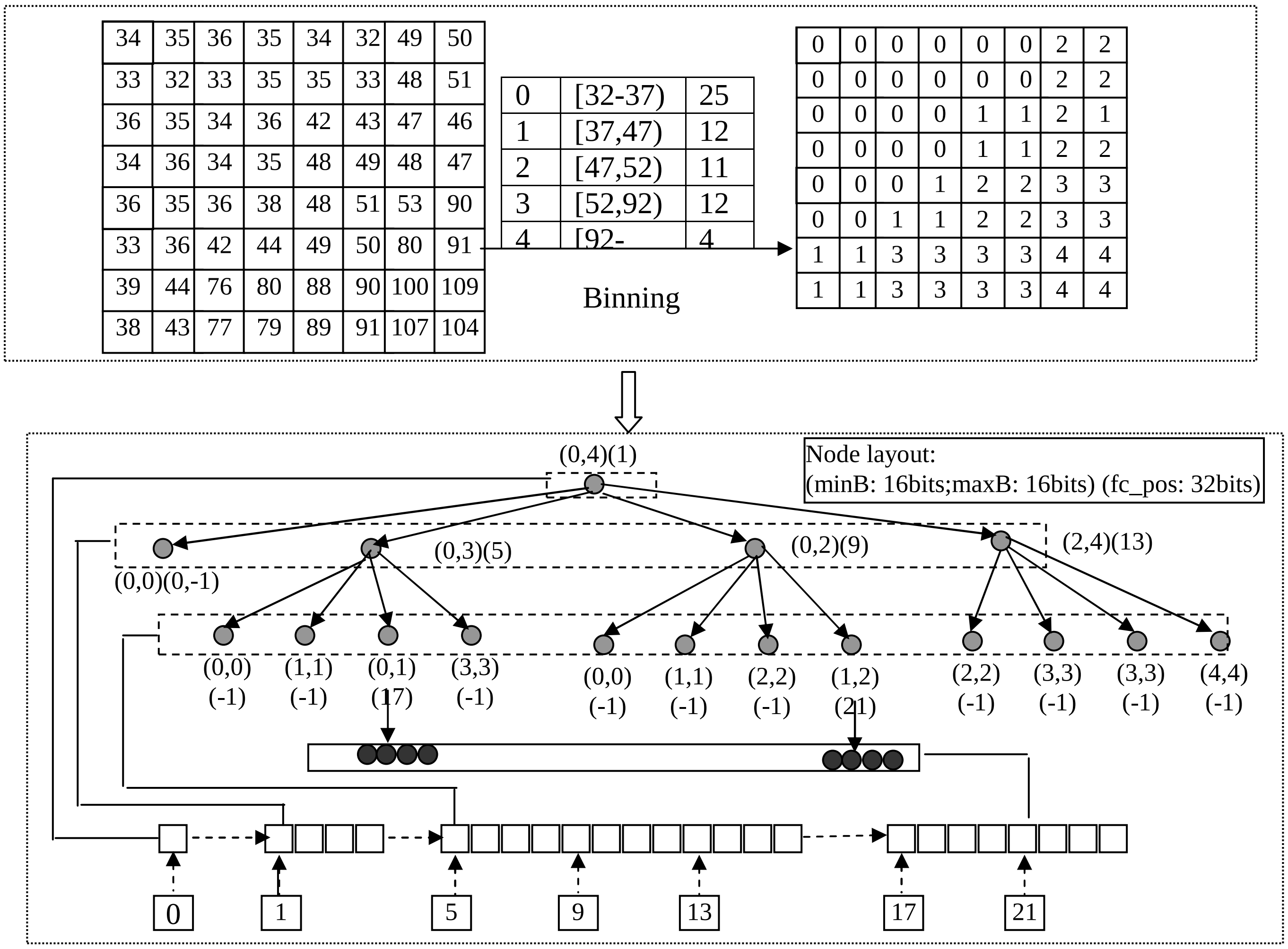
Techniques such as linear quadtrees (Samet 1984) have been developed to externalize main-memory based quadtrees and make them disk-resident. Linear quadtrees can be used to support certain types of queries on top of B+-Tree (Tzouramanis et al. 1998, Aboulnaga and Aref 2001, Manolopoulos et al. 2001). A recent work on managing large-scale species distribution data (Zhang et al. 2009a) associates a set of species identifiers with linear quadtree nodes and uses the PostgreSQL LTREE module[17] to perform window queries by coordinating both the query client and the database server. A main-memory implementation has improved query performance by 2-3 orders as reported in (Zhang 2012, also see online demo at[18]). A Binned Min-Max Quadtree (BMMQ-Tree) data structure that associates min/max statistics of raster cells of a quadrant to the corresponding quadtree node to speed up processing of certain types of queries in a Web environment has been developed (Zhang and You 2010a). BMMQ-Tree is a CPU main-memory indexing structure constructed through a recursive procedure based on the classic Depth First Search (DFS) traversal schema. More recently, the BMMQ-Tree construction algorithm has been implemented on Nvidia GPUs using CUDA directly (Zhang et al. 2010). Unfortunately, the implementation was heavily influenced by its corresponding serial algorithm and the implementation did not fit GPU hardware architecture and GPGPU parallel programming model very well. As a result, as shown in the experiment section, while the implementation was significantly better than the serial CPU implementation (Zhang et al 2010), our new design and implementation is able to achieve another 11X speedup which brings a total speedup of 90X when compared with a new optimized serial CPU implementation. We next introduce the BMMQ-Tree indexing structure and its high level design before we present our parallel primitives based implementation in Section 4.

3 BMMQ-Tree: Data Layout and Parallel Construction Design

The Binned Min-Max Quadtree (BMMQ-Tree) (Zhang and You 2010a, Zhang et al. 2010) can be considered as a special type of quadtree where statistics (minimum and maximum values in this case) are associated with quadtree nodes and the raster cell values are binned to enhance spatial homogeneity and reduce tree complexity. The BMMQ-Tree node layout in the original CPU-based design has been adapted to GPGPUs by replacing four pointers to four child

[16] http://docs.oracle.com/html/B10827_01/geor_intro.htm
[17] http://www.postgresql.org/docs/9.1/static/ltree.html
[18] http://geoteci.engr.ccny.cuny.edu/geoteci/SPTestMap.html

nodes with an array index position to point to the first child node. The layout of the BMMQ-Tree data structure is illustrated in Fig. 2 based on (Zhang et al. 2010). A BMMQ-Tree node has a data field and a position field. The data field, while only the minimum (*minB*) and maximum (*maxB*) values of the raster cells under the node is currently recorded for a BMMQ-Tree, in principle, can store any statistical values, such as mean and deviation. The position field (*fc_pos*) stores the starting position of the first child node in the data stream that holds all the tree nodes linearly based on a Breadth First Search (BFS) tree traversal. For the example shown in Fig. 2, the *minB* and *maxB* values of the root node are 0 and 4 and the first child node position is 1 which indicates that the four children of the root node can be located in the data stream at the positions 1, 2, 3 and 4, respectively. This has been illustrated in the lower part of Fig. 2. As discussed in (Zhang et al. 2010), the BMMQ-Tree structure is cache conscious since sibling nodes are consecutive in the data stream and they are likely to be fetched together into hardware cache lines. The quadtree data structure also has a small memory footprint as only the position of the first child node, instead of the four pointers to all child nodes, is stored. More importantly, the quadtree data structure is GPU-friendly as the data stream of quadtree nodes can be easily held in a 1D array and transferred back and forth between CPU and GPU memories (as well as between disks and CPU memories) without serialization.



34	35	36	35	34	32	49	50
33	32	33	35	35	33	48	51
36	35	34	36	42	43	47	46
34	36	34	35	48	49	48	47
36	35	36	38	48	51	53	90
33	36	42	44	49	50	80	91
39	44	76	80	88	90	100	109
38	43	77	79	89	91	107	104

0	[32-37)	25
1	[37,47)	12
2	[47,52)	11
3	[52,92)	12
4	[92-	4

0	0	0	0	0	0	2	2
0	0	0	0	0	0	2	2
0	0	0	0	1	1	2	1
0	0	0	0	1	1	2	2
0	0	0	1	2	2	3	3
0	0	1	1	2	2	3	3
1	1	3	3	3	3	4	4
1	1	3	3	3	3	4	4

Fig. 2 A BMMQ-Tree Example Illustrating Data Layout (Zhang et al 2010)

Once a BMMQ-Tree is constructed, it can be used to support quite a few types of geospatial queries, such as spatial range queries and Region-of-Interest (ROI) type of queries, for large-scale raster geospatial data. A spatial range query (or window query) returns all spatial objects (including quadrants) that fall within a spatial query window. The properties or the values of the spatial objects then can be retrieved based on the object identifiers. The ROI-type query returns all spatial objects (including quadrants) that satisfy one or more value range criteria, e.g., temperature between [t1,t2) and precipitation between [p1,p2). More generally, given a set of rasters representing environmental variables $\{F_i|0<i<n\}$ over a spatial domain *D* whose value ranges are $\{V_i^h\}$ and $\{V_i^L\}$, respectively, a ROI-type query Q identifies regions in *D* whose cells C_j satisfy the compound condition $\{C_j \mid V_{1j} \in [V_1^{QL}, V_1^{QH}) \text{ op } V_{2j} \in [V_2^{QL}, V_2^{QH}) \text{ op} \ldots \text{op } V_{kj} \in [V_k^{QL}, V_k^{QH})$ where **op** can be either conjunctive and disjunctive and 0<=k<n. V_i^{QL} and V_i^{QH} represent the lower and high bounds of query *Q* for variable *i* (Zhang and You 2010a). It can be seen that a spatial range query maps spatial objects to values while a ROI-type query maps values to spatial objects and they are complementary to each other. Cascading these two types of queries on multiple rasters allows users to identify interesting patterns, such as patterns related to the spatial distributions of geospatial phenomena with certain value thresholds, e.g., storms with precipitation greater than 10mm), and, potential casual relations between multiple rasters through co-location analysis e.g., biodiversity decrease and deforestation/climate changes across the globe (Zhang and You 2010a). A Web-based system has been developed to demonstrate the ROI-type query using the BMMQ-Tree indexing (for single raster only) and can be accessed online[19]. More details on system development are reported in (Zhang and You 2010b).

The high-level design of the parallel construction of a BMMQ-Tree is similar to that has been proposed in (Zhang et al. 2010a) with one key difference. For the sake of clarity, we will introduce the overall design before we discuss the new improvement. There are four steps in the high-level design. The first step transforms a row-major ordered input raster grid (2D array) into a Z-ordered (Morton 1966) 1D array after binning each grid cell in the 2D array. In the second step, for every four consecutive Z-ordered raster cells, an entry is created by recording the minimum and maximum values of the raster cells (i.e., *minB* and *maxB* in Fig. 2). Clearly the resulting entries stored in an array (hereafter referred as the min-max table) also follow Z-order. The third step is to derive higher levels of min-max tables from lower level ones by following the similar procedure in Step 2. Conceptually all the min-max tables at all levels form a pyramid. They can be concatenated into an array from top level to the bottom level with elements at each level follow the Z-order for easy data manipulations. Similarly the numbers of child nodes in the corresponding raster quadrants can also be counted and concatenated as a single array during the two steps where 0 indicates a uniformly distributed quadrant (no child nodes) and 4 otherwise. The fourth step actually calculates the positions of the first child nodes (*fc_pos*) and assembles *minB*, *maxB* and *fc_pos* values into the corresponding quadtree nodes. Finally the quadtree nodes are pruned and connected through *fc_pos* values, which are functionally equivalent to pointers in tree structures on CPU, to complete the construction of a BMMQ-Tree.

While Steps 1-3 are relatively straightforward with the help of the illustrative example in Fig. 3, Step 4 needs more detailed explanation and it is divided into two sub-steps (4.1 and 4.2) as illustrated in Fig. 4 for this purpose. Step 4.1 takes the numbers of child nodes for all quadrants at all levels as the input array (*NumChildren*) and apply an exclusive scan parallel primitive (c.f. Section 2.2) to compute the positions of the first child nodes of the respective

[19] http://geoteci.engr.ccny.cuny.edu/rasterexplorer/comgeotiling/TestOverlay.html

quadrants (stored in *FC_Pos* array). The first child position of the root node (at the position 0) is always 1. As such, the scan should start at the second element of the input array and takes 1 as the initial value. The computed positions (*FC_Pos* array) and the min-max values (*MinmaxTable* array) are assembled to generate the intermediate quadtree nodes (*QuadTree* array) by using the *NumChildren* vector as a stencil to determine how to modify the *fc_pos* values. The rule is that if the element value in the *NumChildren* array is 0, then the corresponding *fc_pos* value of the quadtree node in the *QuadTree* array will be set to -1 to indicate that the quadrant that the quadtree node represents is uniform and no subdivision is needed. In this case, no children nodes exist for the node. In step 4.2, the following approach is used to prune quadtree nodes that represent uniformly distributed quadrants. For each quadtree node (except the root node), in parallel, we extract the number of children nodes of its parent node in the *NumChildren* array. For the ith quadtree node, the position of its parent node in the *NumChildren* array can be simply calculated as (*i*-1)/4 as the intermediate quadtree node array is a full quadtree (pyramid). If both the parent node and the node being examined itself have 0 children, then the node being examined should be pruned. This is because the quadrant that the node being examined represents is part of a larger uniformly distributed quadrant that the parent node represents. We note that if the parent node has 0 children then the node being examined must also have 0 children based on the procedures described in Steps 2 and 3. On the other hand, when the parent node has four children but the node being examined is a leaf quadtree node, i.e., the corresponding value in the *NumChildren* is 0, the node being examined should not be pruned.

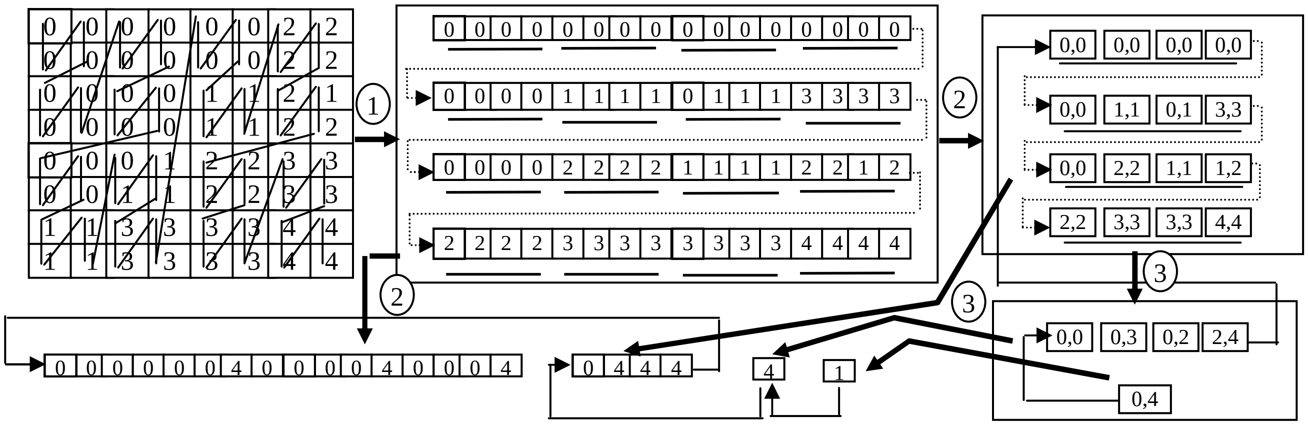


Fig. 3 Illustration of the First Three Steps in BMMQ-Tree Construction

The correctness of the new BMMQ-Tree construction approach can be verified by examining the parent-child relationships of the quadrants in the example dataset in Fig. 4 and the values of the *fc_pos* field of the quadtree nodes in the *QuadTree* array. For example, the value of the *fc_pos* field of the root node is 1 which indicates that the first child node of the root node should be the 1st element in the *QuadTree* array, which is true. As another example, *fc_pos* of the 4th quadtree node in the *QuadTree* array (base 0) is 13 and it is easy to verify that the 13th quadtree node in the *QuadTree* array is indeed the first child node of the 4th quadtree node in the array. We note that for the two elements in the *QuadTree* array that are bolded (the 7th and the 12th element, respectively), they represent the 3rd level quadrants and correspond to the non-shaded quadrants illustrated in the bottom part of Fig. 4. Clearly the 17th-20th elements in the array are the child nodes of the 7th element and the 21st-24th elements are the child nodes of the 12th element, respectively. The data layouts of the 17th-24th elements are not shown in Fig. 4 due to space limit. We also would like to draw attention on how the four uniformly distributed quadrants in the shaded rectangle (with relevant values in the *MinmaxTable*, *NumChildren* and *FC_Pos* arrays) at the top part of Fig. 4 are consolidated into one quadtree node which is the first element (base 0) in the *QuadTree* array. After the *NumChildren* array is derived in Steps 2 and 3, the values of the relevant elements at L2 (Level 2) and L3 (Level 3) are all 0s and they do not contribute to computing *fc_pos* values of the *FC_Pos* array. During the quadtree node pruning process, except for the highest level node (L2) which is kept as a leaf node, all the nodes below L2 that have 0 child nodes are pruned. As we can see from the example, by keeping the correspondences among the numbers of child nodes, the positions of the first child node and the positions of the quadtree nodes, the parent-(first) child relationship is correctly maintained in the resulting quadtree node array. While the design can be implemented on both serial and parallel machines, it is particularly suitable for parallel implementation using parallel primitives as all the required operations are on 1D arrays and no inter-elements communication are needed.

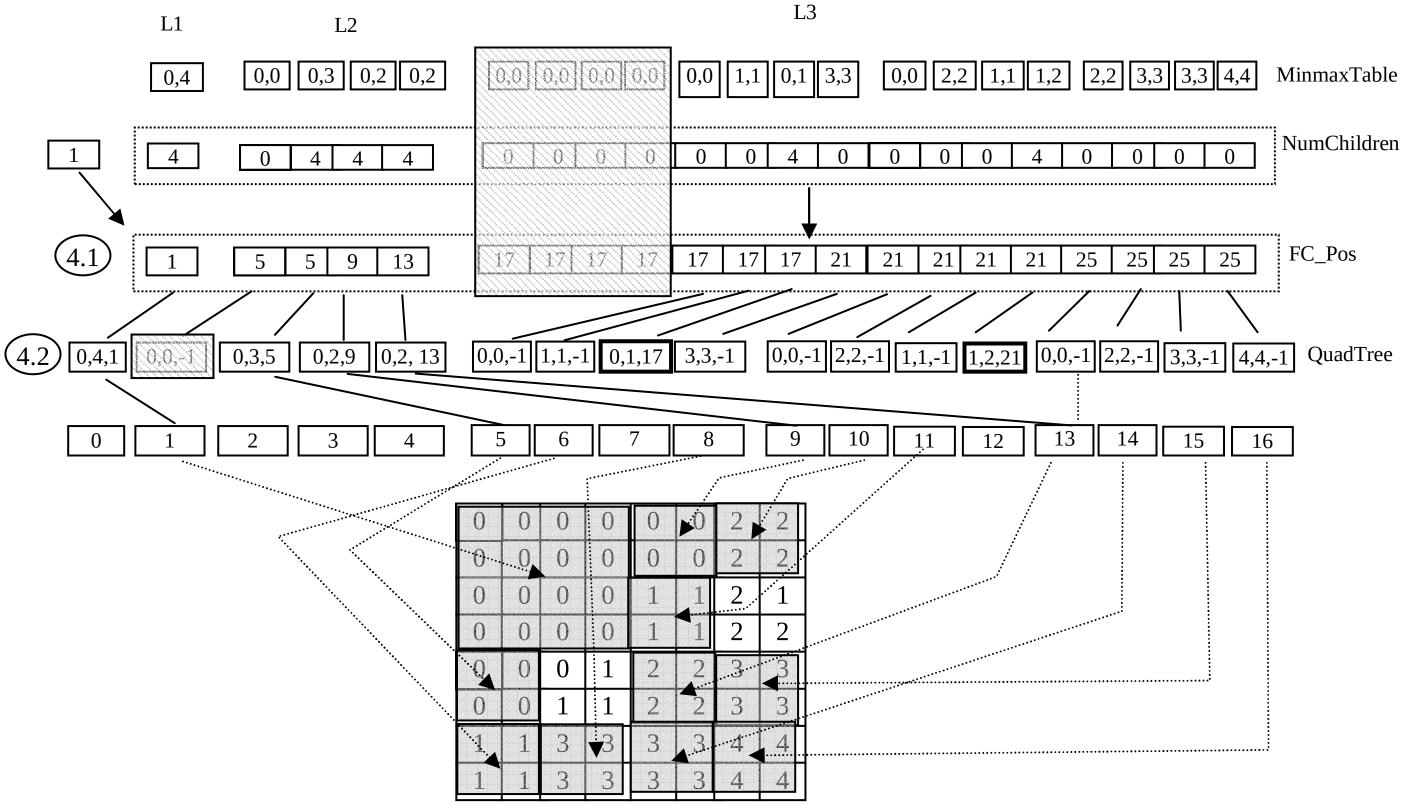


Fig. 4 Illustration of Last Step in BMMQ-Tree Construction

A key difference between the proposed design in this paper and the design reported in (Zhang et al 2010) is that, rather than accessing grid cell values along both row and column dimensions simultaneously as in (Zhang et al 2010), a Z-order transformation is applied right after the binning step and before all the rest of the steps. After the Z-order transformation, the 2D geospatial computing problem is converted into a 1D data processing problem with geospatial semantics embedded. The converted problem is suitable to be solved by chaining a set of parallel primitives that are well understood and efficiently implemented in quite a few parallel libraries. As shown in Section 5, a higher speedup can be achieved by using the optimized implementations of the parallel primitives without requiring deep knowledge of GPU hardware details and outstanding parallel programming skills. Despite the fact that it is still non-trivial to implement the conceptual design using existing parallel primitives, based on our experiences, the technical barriers are significantly lowered.

```
template <typename T>
struct minmax_pair
{
   T min_val;
   T max_val;
   uchar num_children;
};
```

```
template <typename T>
struct quad_node
{
   T min_val,max_val;
   int  fc_pos;
};
```

```
XTOT=4096, YTOT=4096, M=12
blen=pow(4.0f,M)-1)/3)
1) thrust::device_vector<uchar> d_data;
2) device_vector<minmax_pair<uchar> >
minmax_table (blen);
3) device_vector<quad_node<uchar> >
quadtree(blen);
```

Step 0: Bin the raw grid cell values in *r_data* using **transform** and store the results as *b_data*.
Step 1: Convert *b_data* from row-major order to Z-order using **scatter** and store the results in *d_data*
Step 2: Extract the min/max values and number of child quadrant from *d_data* using **transform** and store the results in *minmax_table* and *NumChildren* starting at position *l_p*=(pow(4.0, M-1)-1)/3. Note that 4^{M-1} min-max pairs are generated out of the 4^M grid cells at the level M-1.
Step 3: For *k* from *M*-2 down to 0 (inclusive)

> 3.1 Calculate the starting position and size of the level *k* min-max table: *k_p*=pow((4.0, k)-1)/3 and *k*_s=pow(4.0,*k*)
> 3.2 Extract min/max values and number of child quadrants from *minmax_table* using **transform** and store the results in *minmax_table* and *NumChildren* starting at position *k_p*. Note that *k_s* min-max pairs at the level k are generated from 4**k_s* min-max pairs at the level *k*+1.

Step 4: Extract the numbers of child quadrants from *minmax_table* using **transform** and store it in *NumChildren.*
Step 5: **Exclusive scan** on *NumChildren* with initial value of 1 and store the results in *FC_Pos*.
Step 6: Assemble *minmax_table, NumChildren* and *FC_Pos* into *QuadTree* by using **transform** and store the results in *QuadTree*. The *fc_pos* field is set to -1 if the corresponding value in *NumChildren* is 0.
Step 7 Prune *QuadTree* using **remove_if** by setting the pruning criteria to that both the node being examined and its parent node should have 0 child nodes.

Fig. 5 BMMQ Tree Construction Procedure and Key Data Structures

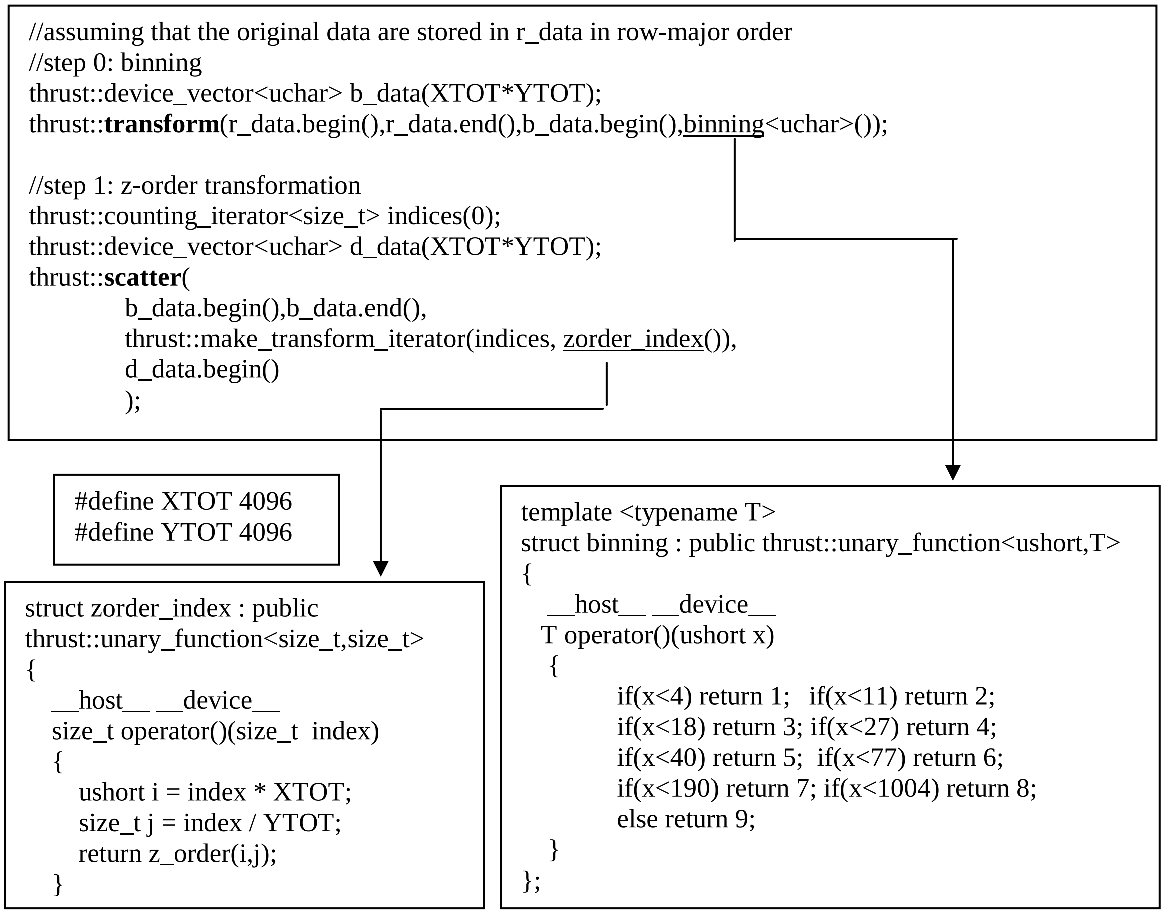
With respect to the time complexity of the proposed BMMQ-tree construction approach, as all steps are linear with respect to the number of grid cells (assuming *n*) in the relevant arrays, the overall time complexity is thus O(*n*). For rasters that have a same dimension, the runtimes are largely input independent on a same hardware configuration. The major workloads to construct the min/max table, computing first child node positions and assembling quadtree nodes remain the same for any input rasters that have a same raster grid dimension. The differences for the pruning step are relatively insignificant. In other words, the performance of the design and implementation is largely data independent, a feature that is desirable in many practical applications.

4 Constructing BMMQ-Trees Using Parallel Primitives

The definitions of major data structures and the overall procedure of the implementation are listed in Fig. 5. The data structures are straightforward translations of the conceptual design in Fig. 2. For the tree construction procedure, roughly speaking, Steps 1-3 in Fig. 5 correspond to Steps 1-3 in the conceptual design (Fig. 3). Similarly Steps 4, 5 and 6 in Fig. 5 correspond to Step 4.1 in Fig. 4, and, Steps 7 in Fig. 5 corresponds to Step 4.2 in Fig. 4, respectively. All the eight steps (including Step 0 for binning) listed in Fig. 5 can be implemented by a call to a parallel primitive including *transform*, *scatter*, *exclusive_scan* and *remove_if* introduced in Section 2.2.

From Fig. 5 we can see that the implementation and the conceptual design match each other very well except that a single step in the conceptual design may require multiple primitives. The overall procedure is pretty straightforward, especially for those who have Thrust and/or STL programming experiences. In general, we consider using parallel primitives (instead of native programming languages) allow us to focus more on high-level designs (e.g., transforming multi-dimensional geospatial computing problems into one-dimensional ones) rather than being buried in details of hardware architectures and programming models. The tradeoffs between coding complexity and code efficiency (Bell and Hoberock 2011) will be further discussed.

Although the implementation is based on the parallel primitives provided by the Thrust library that comes with CUDA SDKs, we believe it is portable to other parallel libraries on both GPUs and multicore CPUs which is left for our future work. While it is beyond the scope of this paper to go through the details of all the parallel primitive invocations in the implementation, we would like to take a few steps as examples to illustrate how the quadtree construction is being implemented using GPU parallel primitives. We also refer to the companying source code package [20] for the details on the rest of the steps in the implementation, including both invocation syntax and the associated functors (C++ function objects).



```
//assuming that the original data are stored in r_data in row-major order
//step 0: binning
thrust::device_vector<uchar> b_data(XTOT*YTOT);
thrust::transform(r_data.begin(),r_data.end(),b_data.begin(),binning<uchar>());

//step 1: z-order transformation
thrust::counting_iterator<size_t> indices(0);
thrust::device_vector<uchar> d_data(XTOT*YTOT);
thrust::scatter(
            b_data.begin(),b_data.end(),
            thrust::make_transform_iterator(indices, zorder_index()),
            d_data.begin()
            );
```

```
#define XTOT 4096
#define YTOT 4096
```

```
struct zorder_index : public
thrust::unary_function<size_t,size_t>
{
   __host__ __device__
   size_t operator()(size_t  index)
   {
      ushort i = index * XTOT;
      size_t j = index / YTOT;
      return z_order(i,j);
   }
```

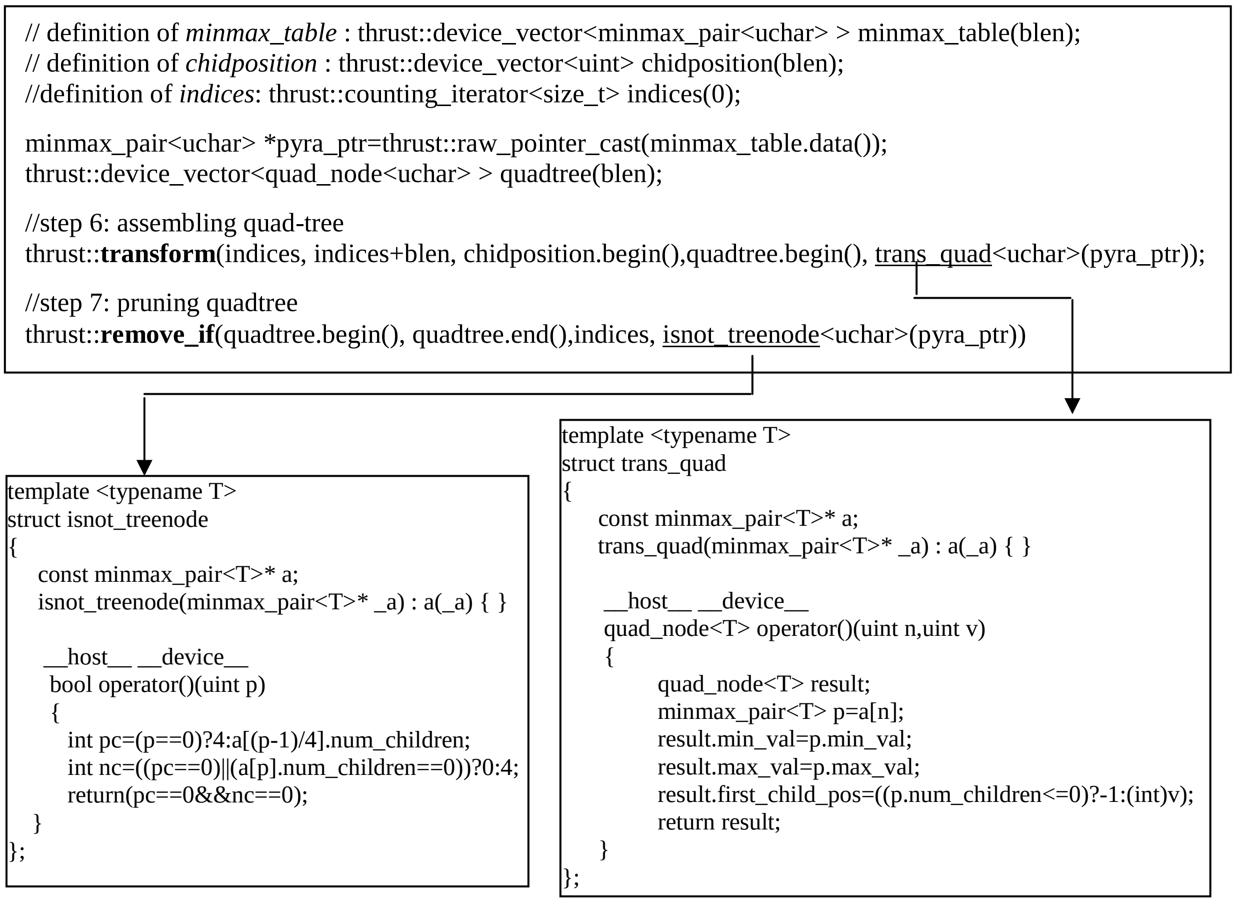
```
template <typename T>
struct binning : public thrust::unary_function<ushort,T>
{
    __host__ __device__
   T operator()(ushort x)
    {
          if(x<4) return 1;   if(x<11) return 2;
          if(x<18) return 3; if(x<27) return 4;
          if(x<40) return 5;  if(x<77) return 6;
          if(x<190) return 7; if(x<1004) return 8;
          else return 9;
    }
};
```

Fig. 6 Code Segment to Illustrate the Binning and Z-order Based Transformation Steps

[20] http://geoteci.engr.ccny.cuny.edu/primquad/primquad.htm

The syntax of invoking the *transform* and the *scatter* primitives and the implementations of the binning and the Z-order transformation functors for Step 1 and Step 2 are illustrated in Fig. 6. We can see that, invoking parallel primitives on GPUs is very similar to calling STL functions which can significantly flatten the learning curve of GPU programming. We also note that iterators and functors are extensively used in the primitives. The *transform* and the *scatter* primitives that are used in these two steps apply the respective functor (*binning* and *zorder_index*) to each element in the input array(s) to produce output arrays, in parallel. In general, the generic parallel primitives that are designed for 1D arrays have excellent scalability and can be realized in multiple parallel hardware architectures, including GPUs. By separating application logic (which is implemented in the *binning* and *zorder_index* functors in the examples) and hardware specific parallel invocations (CUDA kernels to implement the generic parallel primitives), a high level abstraction can be achieved which facilitates productivity of development and portability among different hardware platforms significantly. The *binning* functor takes a 16-bit grid cell value as the input from the *r_data* array and generates an 8-bit bin value to output to the *b_data* array. Each processing unit (e.g. a thread) invokes the *binning* functor independently without communicating with other processing units which is a fundamental requirement of using parallel primitives. The *zorder_index* functor, which transforms a grid from the row-major order to the Z-order within a *scatter* primitive, follows the same schema although it looks a little more complex. Basically the functor takes a row-majored 1D array sequence of a 2D raster grid cell array as the input, calculate the row and column numbers and invoke a function to compute a Morton code (see details in Raman and Wise 2008). The *scatter* primitive embeds the functor into an iterator to generate a Morton code of a position *p* in the sequence of 0..*XTOT*YTOT*-1 (dynamically generated by a *counting_iterator*) and uses the Morton code as the destination position in the output *d_data* array for the element at position *p* of the input array *b_data*. Essentially the line of code is a combination of a *scatter* primitive and a *transform* primitive. The combination successfully avoids outputting the computed Morton codes to an array in GPU device memory and reading them back to registers later, an optimization technique that is desirable.

Similarly the syntax of invoking the *transform* and the *remove_if* primitives and the implementations of assembling quadtree nodes and pruning the quadtree functors for Step 6 and Step 7 are illustrated in Fig. 7. Note that *indices* is a *counting_iterator* variable that has been defined previously which can serve as array subscripts in many applications. Also note that *minmax_table* and *chidposition* arrays are filled in Steps 2/3 and 4/5, respectively (they are omitted here due to space limit). We would like to draw attention on the constructors of the *trans_quad* and *isnot_treenode* functors where the pointer pointing to the first element of the *minmax_table* is passed to the two functors so that the functors can access any elements in the *minmax_table* array when they are invoked. The position of the element that is being processed by a processing unit is passed to the *operator* function of both of the functors (*n* for *trans_quad* and *p* for *isnot_treenode*, respectively) so that the respective *operator* function can decide what to return based on the elements in *minmax_table* that are relative to the position value and arguments that are being passed to the *operator* function. For example, in *isnot_treenode*, the number of child nodes of both the parent node and the node being examined, i.e., nodes at (*p*-1)/4 and *p*, are taken into consideration. The implementations of the two functors follow the design in Section 3 very well and we left the verification to readers. Note that the *transform* primitive used in Step 6 has a more complex form than the one used in Step 2 of Fig. 6. Here two input arrays (*indices* and *chidposition*) are used. As a consequence, the *trans_quad* functor takes two parameters in its *operator* function with each extracted from the respective input array. In contrast, the *binning* functor in Fig. 6 takes only one parameter in its *operator* function. In general, our implementation aims at making full use of the powerful yet flexible primitives based programming framework that the Thrust library has provided. We believe a similar architecture design can be adopted for developing a geospatial specific parallel primitive library which is one of our long term goals.



```
// definition of minmax_table : thrust::device_vector<minmax_pair<uchar> > minmax_table(blen);
// definition of chidposition : thrust::device_vector<uint> chidposition(blen);
//definition of indices: thrust::counting_iterator<size_t> indices(0);

minmax_pair<uchar> *pyra_ptr=thrust::raw_pointer_cast(minmax_table.data());
thrust::device_vector<quad_node<uchar> > quadtree(blen);

//step 6: assembling quad-tree
thrust::transform(indices, indices+blen, chidposition.begin(),quadtree.begin(), trans_quad<uchar>(pyra_ptr));

//step 7: pruning quadtree
thrust::remove_if(quadtree.begin(), quadtree.end(),indices, isnot_treenode<uchar>(pyra_ptr))
```

```
template <typename T>
struct isnot_treenode
{
    const minmax_pair<T>* a;
    isnot_treenode(minmax_pair<T>* _a) : a(_a) { }

     __host__ __device__
      bool operator()(uint p)
      {
        int pc=(p==0)?4:a[(p-1)/4].num_children;
        int nc=((pc==0)||(a[p].num_children==0))?0:4;
        return(pc==0&&nc==0);
    }
};
```

```
template <typename T>
struct trans_quad
{
     const minmax_pair<T>* a;
     trans_quad(minmax_pair<T>* _a) : a(_a) { }

      __host__ __device__
      quad_node<T> operator()(uint n,uint v)
      {
              quad_node<T> result;
              minmax_pair<T> p=a[n];
              result.min_val=p.min_val;
              result.max_val=p.max_val;
              result.first_child_pos=((p.num_children<=0)?-1:(int)v);
              return result;
      }
};
```

Fig. 7 Code Segment to Illustrate Assembling Quadtree Nodes and Pruning Quadtree Using Parallel Primitives

5 Experiments and Results

We use the same global 30-arcsecond January Precipitation dataset from WolrdClim website[21] that has been used in (Zhang and You 2010a, Zhang and You 2010b and Zhang et al 2010) for CPU and GPU based implementations. Since the dataset was divided into 4096*4096 tiles in (Zhang et al 2010), we apply the same tiling schema in this study. As the valid values for raster cells range from 0 to 1004, we have used 8 bines with bin boundaries at (0, 4, 11, 18, 27, 40, 77, 190, 1004). All experiments are performed on a Dell T5400 machine with dual quad-core Intel Xeon E5405 CPUs (2.00 GHz, only one core is used for the experiments) and an Nvidia Quadro 6000 GPU card[22] with 448 cores and 6 gigabytes global memory. CUDA SDK 4.1 and Thrust 1.6 are used. We have experimented on multiple 4096*4096 tiles (32 MB chunks for 16-bits rasters) from the global 30-arcsecond January Precipitation dataset. As discussed in Section 3, the performance is largely data independent for rasters with a same dimension and the

[21] http://biogeo.ucdavis.edu/data/climate/worldclim/1_4/grid/cur/prec_30s_bil.zip
[22] http://www.nvidia.com/object/product-quadro-6000-us.html

runtimes increase linearly with respect to the numbers of grid cells in rasters. This has been verified by experimenting on a 16384*16384 raster which requires almost exactly 16 times runtime. As such, we will restrict our discussions of experiment results on 4096*4096 tiles.

We believe the title size with a dimension of 4096*4096 is suitable for most GPU devices that have 256 MB or above graphics memory. By optimizing memory utilization in our current primitives based implementation, it is possible to accommodate for GPU devices with smaller memory capacities and we leave it for future work. In this section, we focus on the comparisons among GPU and CPU based implementations with different strategies to understand the realized and potential performance gains from GPU hardware and GPGPU technologies. We encourage readers to download our source code package and test the performance of the implementations (the URL has been provided previously). Instructions on using tools to extract raw data from arbitrary rasters or images, programs for chunking the raw data into tiles with widths and heights (2's powers, required by the implementations) and suggestions on defining bin boundaries have also been provided.

The runtime of our primitives based GPU implementation (hereafter referred as GPU-Primitive) is 13.33 milliseconds and we use it as the baseline for comparison purposes. We have re-implemented the classic recursive Deepest-First Search (DFS) based serial CPU implementation by adopting a few performance optimization techniques for fair comparisons. The implementation is referred as CPU-DFS. Additionally, we have implemented the new design proposed in this paper on CPUs using a single processor to loop through all the elements in the respective array to simulate the parallel execution. We refer the implementation as CPU-SIM. The CUDA based implementation reported in (Zhang et al 2010) is tested without any changes and is referred as GPU-OLD. The two CPU implementations are complied with -O2 flag for optimizations to ensure fair comparisons. The experiment results show that the runtimes of CPU-DFS, CPU-SIM and GPU-OLD are 1191.87 milliseconds, 1044.36 milliseconds and 147.23 milliseconds, respectively, for multiple 4096*4096 raster tiles on average.

When compared the GPU-Primitive implementation against the rest three implementations, a speedup of 89.4X over CPU-DFS, 78.34X over CPU-SIM and 11.0X over GPU-OLD has been achieved. We attribute the 11.0X speedup over GPU-OLD to better coalesced memory accesses due to Z-order transformation and implicitly use of shared memory for scan. The signficant 89.4X speedup over CPU-DFS is due to the excessive dynamic memory allocation and de-allocation and cache unfriendly data accesses in the CPU-DFS implementation. It is a little surprising that CPU-SIM does not significantly outperform CPU-DFS. We had expected that CPU-SIM would be significantly better than CPU-DFS because the dynamic memory management overheads in CPU-DFS were largely removed and arrays are cache friendly in most of the steps (except the Z-order transformation step) in the CPU-SIM implementation. Unfortunately this is not true. Further analysis has revealed that the binning and the Z-order transformation (combined in CPU-SIM) took the majority of the runtime (974.92 milliseconds) while all the rest steps combined took only 69.44 milliseconds. In contrast to the combined runtime of the same binning and Z-order transformation steps in GPU-primitive which is only 4.40 milliseconds, an impressive 260X speedup has been observed. We suspect that the cache unfriendly nature of Z-order transformation process on CPUs can be the performance bottleneck which requires further investigation. GPU-Primitive still gains about 7.8X speedup, which is calculated as 69.44/(13.33-4.40), for the rest of the steps. Both results have demonstrated the advantages of GPGPU technologies for geospatial computing, which is often both data and computing intensive, by leveraging the large numbers of processing cores and high memory bandwidths available on GPU devices when compared with CPUs.

The experiment results have positive implications for indexing and querying large-scale rasters. Experiments for 16-bit rasters using both 4096*4096 tiles (13.33 milliseconds) and 16384*16384 (230.75 milliseconds) tiles have suggested an indexing rate of more than 1.25 billion cell/pixel per second (2^{30}/s). The processing rate is lower but comparable to PCI-Express[23] data transfer rate between the CPU and GPU memories on our machine which means that the sustainable processing rate is achievable when interleaving data transfer and processing. The processing rate suggests that the data generated by GOES-R satellites at the global scale each day, i.e., 288 coverages and 16 bands with each coverage having approximate 1/4 billion pixels can be indexed in less than 20 minutes[24] on a single GPU device. Although many applications require more sophisticated computations than constructing BMMQ-Trees, GPGPU computing seems to be a cost-effective way for large-scale geospatial computing. Furthermore, as Nvidia Kepler GPUs[25] that are currently available on the market have more than 3000 cores and PCI-Express 3 standard allows up to 16 GB/s data transfer among CPUs and GPUs, we expect that the achievable data processing rate can be further improved on commodity GPU devices. Although it is unlikely that disk I/O speed can reach a 2 GB/s rate any time sooner to match the 1 billion raster cells per second GPU processing rate on a personal workstation, we argue that this is quite possible in a cluster computing environment where parallel file systems are available. Efficiently streaming large-scale raster data from disks to CPU memoires and to GPU memories as well as utilizing parallel file system to further speed up realizable processing rate are left for our future work.

Conclusion and Future Works

In this study, we have adopted a transformation based approach to effectively and efficiently utilizing massively data parallel GPGPU technologies for geospatial computing. By ordering grid cells of geospatial rasters based on Z-order, we transform a multi-dimensional geospatial indexing (BMMQ-Tree construction) problem into a set of smaller problems with each can be solved by using a generic parallel primitive optimized for one-dimensional arrays on GPGPUs. Our experiments have shown that the primitive based GPU implementation on an Nvidia Quadro 6000 GPU device has achieved nearly 90X speedup over an optimized serial CPU implementation and is 11X faster than a previous GPU implementation. We believe the approach can be extended to a large family of geospatial computing problems by designing proper transformation schemas. Our additional research efforts along the direction, such as constructing DEMs from large-scale point datasets (You and Zhang, 2012) and several spatial join processing on vector geospatial data (Zhang and You 2012, Zhang et al. 2012a, Zhang et al. 2012b), seem to be encouraging. These research and development efforts can also serve as case studies towards developing high performance parallel geospatial computing primitives to bridge between conceptual deigns of geospatial computing models, software developments and hardware parallel executions.

There is plenty of room for future work. First of all, we would like to extend the quadtree based indexing to include query processing on GPGPUs, e.g., spatial range queries, ROI-type queries and spatial joins on both raster and vector geospatial data. Second, although we have been using a single GPU device for our data structure and algorithm development in a personal computing environment, we plan to extend the approach to a cluster computing environment

[23] http://en.wikipedia.org/wiki/PCI_Express
[24] Approximately calculated as |W|*|H|*|T|*|B|/R=(360*60)*(180*60)*(12*24)*16/(2^30)=1001sec =16.7min
[25]http://www.nvidia.com/object/nvidia-kepler.html

using grid/cloud computing resources to further test the scalability of the proposed approach. Finally, we have strong interest in developing geospatial specific parallel primitives to support large-scale geospatial computing in a cyberinfrastructure framework with respect to open source software development and providing services to the user community over the Web.

Acknowledgement
This work is supported in part by PSC-CUNY Grants #65692-00 43 and #66724-00 44 an NSF Grant IIS-1302423.

References

1. Aboulnaga, A. and Aref, W. G., 2001. Window query processing in linear quadtrees. Distributed and Parallel Databases, 10(2), 111-126.
2. Ali, M.H., Saad, A.A. and Ismail, M.A., 2005. The PN-tree: A parallel and distributed multidimensional index. Distributed and Parallel Databases, 17(2), 111–133.
3. Akdogan, A., Demiryurek, U., et al., 2010. Voronoi-Based Geospatial Query Processing with MapReduce. Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom'10), 9-16.
4. Armstrong, M.P., Pavlik, C.E. and Marciano, R., 1994. Parallel-processing of spatial statistics. Computers and Geosciences, 20(2), 91–104.
5. Bell, N. and Hoberock, J., 2011. Thrust: A Productivity-Oriented Library for CUDA. In Hwu, W.-M. W (eds.) GPU Computing Gems: Jade Edition. Morgan Kaufmann.
6. Clematis, A., Mineter M. and Marciano, R., 2003. High performance computing with geographical data. Parallel Computing, 29(10), 1275–1279.
7. Cary, A., Sun, Z., Hristidis, V. and Rishe, N., 2009. Experiences on processing spatial data with MapReduce. Proceedings of the 21st International Conference on Scientific and Statistical Database Management(SSDBM'09), 302-319.
8. Chan, Y. K. and Chang, C. C., 2004. Block image retrieval based on a compressed linear quadtree. Image and Vision Computing, 22(5), 391-397.
9. Chung, K. L., Liu, Y. W., et al., 2006. A hybrid gray image representation using spatial- and DCT-based approach with application to moment computation. Journal of Visual Communication and Image Representation, 17(6), 1209-1226.
10. Cignoni, P., Marino, P., et al., 1997. Speeding up isosurface extraction using interval trees. IEEE Transactions on Computer Graphics, 3(2), 158-170.
11. Dean, J. and Ghemawat S., 2010. MapReduce: a flexible data processing tool. Communications of the ACM 53(1), 72-77.
12. Gaede V. and Gunther O., 1998. Multidimensional access methods. ACM Computing Surveys, 30(2), 170-231.
13. Garland M. and Kirk, D. B., 2010. Understanding throughput-oriented architectures. Communications of the ACM, 53(11), 58-66.
14. Gress, A. and Klein, R., 2004. Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. Graphical Models, 66(6), 370-397.
15. Guan, Q. and Clarke K., 2010. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. International Journal of Geographical Information Science, 24(5), 695-722.
16. Han, S. H., Heo J., et al., 2009. Parallel processing method for airborne laser scanning data using a PC cluster and a virtual grid. Sensors, 9(4), 2555–2573.

17. Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann.
18. Hoel, E.G. and Samet, H., 2003. Data-parallel polygonization. Parallel Computing, 29(10), 1381–1401.
19. Hong, S., Kim, S. K., et al., 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11), 267-276.
20. Hwu, W.-M. W (eds.), 2011a. GPU Computing Gems: Emerald Edition. Morgan Kaufmann
21. Hwu, W.-M. W (eds.), 2011b. GPU Computing Gems: Jade Edition. Morgan Kaufmann
22. Kamel, I. and Faloutsos, C., 1992. Parallel r-trees. Proceedings of the ACM SIGMOD International conference on Management of data (SIGMOD'92), 195–204.
23. Kirk, D. B. and Hwu, W.-M., 2010. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann.
24. Lin, T. W., 1997. Compressed quadtree representations for storing similar images. Image and Vision Computing 15(11), 833-843.
25. Liu, Y., Wu, K., Wang, S. et al., 2010. A MapReduce approach to Gi*(d) spatial statistic. Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems (HPDGIS'10), 11-18.
26. Luo, L., Wong, M. D. F., et al., 2011. Parallel implementation of R-trees on the GPU. Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC), 353-358.
27. Manolopoulos Y., Nardelli, Y., E., et al., 2001. A generalized comparison of linear representations of thematic layers. Data & Knowledge Engineering, 37(1), 1-23.
28. Manouvrier, M., Rukoz, M., et al., 2002. Quadtree representations for storage and manipulation of clusters of images. Image and Vision Computing, 20(7), 513-527.
29. McCool, M., Reinders, J. and Reinders, J., 2012. Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann.
30. Mineter, M., 2003. A software framework to create vector-topology in parallel GIS operations. International Journal of Geographical Information Science, 17(3), 203-222.
31. Molnár, F., T. Szakály, et al., 2010. Air pollution modelling using a Graphics Processing Unit with CUDA. Computer Physics Communications 181(1), 105-112.
32. Morton, G.M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. IBM Technical report.
33. Ortega, L. and Rueda, A., 2010. Parallel drainage network computation on CUDA. Computers and Geosciences, 36(2), 171-178.
34. Oryspayev, D., Sugumaran, R., et al., 2012. LiDAR data reduction using vertex decimation and processing with GPGPU and multicore CPU technology. Computers and Geosciences, 43, 118-125.
35. Patel, J.M. and DeWitt, D.J., 2000. Clone join and shadow join: two parallel spatial join algorithms. Proceedings of the 8th ACM international symposium on Advances in Geographic Information Systems (GIS'00) ,54–61.
36. Qin C.-Z. and Zhan, L., 2012. Parallelizing flow-accumulation calculations on graphics processing units - From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm, Computers & Geosciences, 43, 7-16.
37. Raman, R. and Wise, D.S., 2008. Converting to and from Dilated Integers. IEEE Transactions on Computers, 57(4), 567-573.
38. Samet, H., 2005. Foundations of Multidimensional and Metric Data Structures Morgan Kaufmann Publishers Inc.

39. Samet, H. 1985. Data-Structures for Quadtree Approximation and Compression. Communications of the ACM, 28(9), 973-993.
40. Samet, H., 1984. The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys, 16(2), 187-260.
41. Schmit, T.J., Li, J., et al., 2009. High-spectral- and high-temporal resolution infrared measurements from geostationary orbit. Journal of Atmospheric and Oceanic Technology, 26(11), 2273-2292
42. Steinbach, M. and Hemmerling, R., 2012. Accelerating batch processing of spatial raster analysis using GPU. Computers and Geosciences, 45, 212-220.
43. Theobald, D. M., 2005. GIS Concepts and ArcGIS Methods, 2nd Ed., Conservation Planning Technologies, Inc.
44. Tzouramanis, T., Vassilakopoulos, M,. et al., 1998. Overlapping linear quadtrees: a spatio-temporal access method. Proceedings of the 6th ACM international symposium on Advances in Geographic Information Systems (GIS'98), 1-7.
45. You, S. and Zhang, J., 2012. Constructing natural neighbor interpolation based grid DEM using CUDA. Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications (COM.Geo '12), Article#28, 6 pages.
46. Wang, C. and Chiang Y. J., 2009. Isosurface Extraction and View-Dependent Filtering from Time-Varying Fields Using Persistent Time-Octree (PTOT). IEEE Transaction on Computer Graphics, 5(6), 1367-1374.
47. Wang, S. W. and Liu, Y., 2009. TeraGrid GIScience Gateway: Bridging cyberinfrastructure and GIScience. International Journal of Geographical Information Sciences, 23(5) 631-656.
48. Wang, S. W., Cowles, M. K., et al., 2008. Grid computing of spatial statistics, using the TeraGrid for Gi*(d) analysis. Concurrency and Computation: Practice and Experience, 20(14), 1697-1720.
49. Wang, S.W. and Armstrong, M.P., 2003. A quadtree approach to domain decomposition for spatial interpolation in grid computing environments. Parallel Computing 29(10), 1481–1504
50. Wilhelms, J. and Vangelder, A., 1992. Octrees for Faster Isosurface Generation. ACM Transactions on Graphics, 11(3), 201-227.
51. Xu, X.W., Jager, J. and Kriegel, H.P, 1999. A fast parallel clustering algorithm for large spatial databases. Data Mining and Knowledge Discovery, 3(3), 263–290.
52. Yang, C. W., Goodchild, M. A, et al., 2011. Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing. International Journal of Digital Earth, 4(4), 305-329.
53. Yang, C. W., Raskin, R. and Goodchild, M. A., 2010. Geospatial cyberinfrastructure: Past, present and future. Computers, Environment and Urban Systems, 34(4), 264–277.
54. Zhang, J., 2012. A high-performance web-based information system for publishing large-scale species range maps in support of biodiversity studies. Ecological Informatics, 8, 68-77.
55. Zhang, J. and You, S., 2012. Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial'12), 23-32.
56. Zhang, J., You, S. and Gruenwald, L., 2012a. U^{2}STRA: high-performance data management of ubiquitous urban sensing trajectories on GPGPUs. Proceedings of the 2012 ACM workshop on City data management workshop (CDMW'12), 5-12.
57. Zhang, J., You, S. and Gruenwald, L., 2012b. High-Performance Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. Proceedings of the fifteenth international workshop on Data warehousing and OLAP (DOLAP'12), 89-96.

58. Zhang, J., 2011. Speeding Up Large-Scale Geospatial Polygon Rasterization on GPGPUs. Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems (HPDGIS'11), 10-17.
59. Zhang, J., You, S. and Gruenwald, L., 2011. Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs. Proceedings of the 19th ACM international symposium on Advances in Geographic Information Systems (GIS'11), 457-460.
60. Zhang, J., 2010. Towards personal high-performance geospatial computing (HPC-G): perspectives and a case study. Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems (HPDGIS'10), 3-10.
61. Zhang, J. and You, S., 2010a. Supporting Web-based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree. Proceedings of the 22nd International Conference on Scientific and Statistical Database Management Conference (SSDBM'10), 379-396.
62. Zhang, J. and You, S., 2010b. Dynamic Tiled Map Services: Supporting Query-Based Visualization of Large-Scale Raster Geospatial Data. Proceedings of the 1st International Conference on Computing for Geospatial Research & Application (COM.Geo'10), Article #19, 8 pages.
63. Zhang, J., You, S. and Gruenwald, L., 2010. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. Proceedings of the 18th ACM international symposium on Advances in Geographic Information Systems (GIS'10), 450-453.
64. Zhang, J,, Gertz, M. and Gruenwald, 2009a. Efficiently managing large-scale raster species distribution data in PostgreSQL. Proceedings of the 17th ACM international symposium on Advances in Geographic Information Systems (GIS'09), 316-325.
65. Zhang, S., Han, J., et al., 2009b. SJMR: Parallelizing spatial join with MapReduce on clusters. Proceedings of the IEEE International Conference on Cluster Computing Workshops (CLUSTER '09), 1-8.
66. Zhou, X., Abel, D. J., and Truffet, D., 1998. Data partitioning for parallel spatial join processing. GeoInformatica, 2(2), 175–204.