Simplifying High-Performance Geospatial Computing on GPGPUs Using Parallel Primitives: A Case Study of Quadtree Constructions on Large-Scale Geospatial Rasters

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 resources and the emerging General Purpose computing on GPU (GPGPU) technologies provide an alternative and complementary solution to existing cluster based high-performance geospatial computing. However, the complexities of the unique GPGPU hardware architectures and the steep learning curve of GPGPU programming have imposed signficant technical challenges on the geospatial computing community to develop efficient parallel geospatial data structures and algorithms that can make full use of the hardware capabilities to solve ever growing large and complex real world geospatial problems. In this study, we propose a practical approach to simplifying high-performance geospatial computing on GPGPUs by using parallel primitives. We take a case study of quadtree construction on large-scale geospatial rasters to demonstrate the effectiveness of the proposed approach. Comparing the proposed parallel primitives based implementation with a naïve CUDA implementation, a signficant reduction on coding complexity and a 10X speedup have been achieved. We believe that GPGPU based software development using generic parallel primitives can be a first step towards developing geospatial-specific and more efficient parallel primitives for high-performance geospatial computing in both personal and cluster computing environments and boost the performance of geospatial cyberinfrastructure.

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). While grid and cloud computing are currently the two leading frameworks for high performance geospatial computing in the context of geospatial cyberinfrastructure research and developments, there are increasing interests in GPGPU technologies, i.e., General Computing on Graphics Processing Units, for high-performance geospatial data processing. Indeed, as argued in a position paper (Zhang 2010), high-end workstations equipped with GPGPU devices with hundreds 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]). Besides being cost-effective and energy efficient in solving small to medium sized problems directly on GPU-equipped personal workstations, more importantly, as modern grid

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

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 grid/cloud computing resources on cluster computers to solve larger scale problems.

Geospatial data processing on GPGPUs have attracted signficant research and application interests in the past few years ranging from data management to physics based environmental simulation. The throughput-oriented architecture designs of GPGPUs (Garland and Kirk 2010) 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 global geospatial operations whose parallelism can not be easily mapped to GPGPU computing blocks and threads. 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. Unfortunately, the current generation of GPGPUs has quite different hardware features than CPUs and it is nontrivial to port such algorithms from CPUs to GPUs. Despite signficant research efforts in both geospatial computing and other related domains (e.g., Zhang et al 2010, Zhang et al 2011, Zhou et al 2008, Hou et al 2011, Zhou et al 2011, Luo et al 2011), constructing tree indices on GPGPUs remain difficult by using mainstream programming abstractions (languages, tools et al.), such as CUDA [3] and OpenCL[4].

An alternative approach is to use high level parallel programming libraries or Application Programming Interfaces (APIs) to hide hardware details, simplify coding and improve overall software development efficiency while exploiting the parallel processing power of GPUs. Parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines. The idea is similar to the development of the Standard Template Library (STL[5]) and the Boost C++ libraries[6] on CPUs that have been proven to be successful in many aspects. Quite a few GPGPU based parallel libraries, such as CUDPP[7] and Thrust[8], are currently available and provide efficient implementations of parallel primitives on GPGPUs. While generic parallel primitives have been applied to quite a few domains, including relational data management {He et al 2009), to the best of our knowledge, we are not aware of previous research in using parallel primitives to simplify geospatial indexing in the context of high-performance geospatial computing.

The research reported in this paper complements previous works on constructing an indexing data structure called Binned Min-Max Quadtree (BMMQ-Tree) on large-scale raster geospatial data on both CPUs (Zhang and You 2010a) and GPUs (Zhang et al 2010). The motivation of the research on indexing large-scale rasters, as detailed in (Zhang and You 2010a), was to allow efficient dynamic queries on large-scale rasters for effective visual explorations[9]. The volumes of raster geospatial data are increasing quickly. For example, the next generation

[3] http://www.nvidia.com/object/cuda_home_new.html
[4] http://www.khronos.org/opencl/
[5] http://www.sgi.com/tech/stl/
[6] http://www.boost.org/
[7] http://code.google.com/p/cudpp/
[8] http://code.google.com/p/thrust/
[9] A Web-based system has been developed to demonstrate the feasibility and effectiveness and can be accessed online at http://geoteci.engr.ccny.cuny.edu/rasterexplorer/comgeotiling/TestOverlay.html and more details on system development are reported in (Zhang and You 2010b).

geostationary weather satellite GOES-R serials[10] 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). 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 better understand the environments and their human impacts. A previous implementation on constructing BMMQ-Trees using CUDA programming model has achieved signficant speedups when compared with the serial CPU implementation (Zhang et al 2010). However, due to the complexity of the GPU hardware and the steep learning curve of the CUDA program model, the implementation (hereafter termed as CUDA-Naïve) is far from optimal. Our recent research has shown that re-implementing the BMMQ-Tree construction algorithm using parallel primitives can speed up the new implementation by an order (10X) while significantly reduce coding complexity.

We believe 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 overflowed 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 works. Section 3 reviews the BMMQ-Tree layout and its construction using the CUDA-Naive approach. Section 4 presents the details of BMMQ-Tree constructions on GPGPUs using parallel primitives. Section 5 provides experiment and evaluation details as well as some discussions. Finally Section 6 is the conclusion and future work directions.

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 CPU to accelerate rendering of 3D or 2D graphics. The highly parallel structures of modern GPU devices, such as AMD/ATI Radeon[11] and Nvidia GeForce/Quadro series[12], 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 interests in many scientific research areas 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

[10] http://www.goes-r.gov/
[11] http://en.wikipedia.org/wiki/Radeon
[12] http://developer.nvidia.com/cuda-gpus

deliver equivalent performance at 1/20th of power consumption and 1/10th of cost[13]. As many reasonably current desktop computers have already 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)[14] 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) 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[15] is based on OpenCL 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., 16 or 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., 512M or 4G) 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 device 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 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 architecture, the learning curve of efficient CUDA programming can be steep. The Thrust library 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.

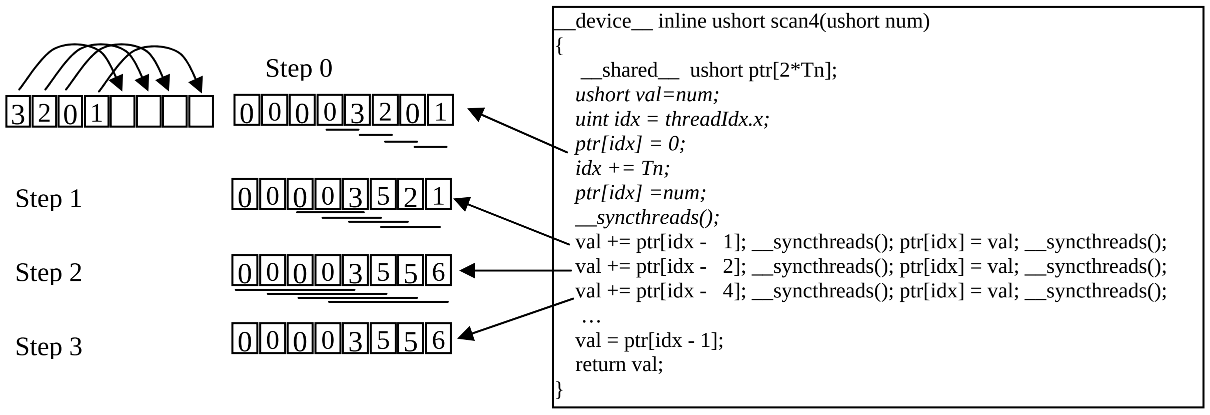
2.2 Parallel Primitives in the Thrust Library

Parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines. The behaviors of popular parallel primitives on vector data are well-understood. 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 the end users and parameters are fine-tuned for typical applications so that users do not need to specify such parameters explicitly. On the other hand, such APIs usually use template or generic based programming[16] techniques so that the same set

[13] http://www.nvidia.com/object/io_1227008280995.html
[14] http://en.wikipedia.org/wiki/SIMD
[15] http://developer.amd.com/sdks/AMDAPPSDK/
[16] http://en.wikipedia.org/wiki/Generic_programming

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 multicore computing platforms. This unique feature further makes parallel primitives based algorithm developments attractive when compared to 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 that we will use in developing our quadtree construction algorithm.

(1) *Scan*. The *Scan* primitive computes the cumulative sum of a vector. Both the inclusive and exclusive scans are available. For example, exclusive_scan([3,2,0,1])$\rightarrow$([0,3,5,5]) while inclusive_scan ([3,2,0,1])$\rightarrow$([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 which is important in our implementation, 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 is 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.



```
__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 A simplify illustration of *Scan* implementation using four threads in CUDA

(2) *Copy* and *Copy_if*. The functionality of the two primitives is self-evident. In this research, we use *Copy* to move groups of entries from one location to another, mostly within a same vector, i.e., in-place copy. The *Copy_if* primitive takes an additional unary function as a parameter to tell weather the corresponding vector element should be copied to the output vector or not.

(3) *Transform*. The basic form of *Transform* applies a unary function to each entry of an input vector and stores the result in the corresponding position in an output vector. Transform is more general than *Copy* as it allows a user defined operation to be applied to vector elements

rather than simply copying. Similar to *Copy_if*, there is also a *Transform_if* primitive where only vector elements are evaluated to true based on a second unary function is transformed using the first unary function.

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

The alert readers many have observed that these parallel primitives work on flat 1D vectors 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 vector based generic parallel primitives as the building blocks to construct parallel geospatial processing modules. On the other hand, the current generation of GPGPU devices work best with flat 1D vectors in many cases. Mapping between multi-dimensional geospatial data to flat 1D vectors can potentially help identifying parallelisms in geospatial computing and facilitate designing more efficient, geospatial-specific data structures and algorithms on GPGPUs for geospatial computing.

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 2011). 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[17]) 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 interests in geospatial data processing using the MapReduce parallel computing framework (Dean and Ghemawat 2010) and the open source Hadoop implementation[18] 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 2009), geostatistics (Liu et al 2010) and nearest neighbor queries on voronoi diagrams (Akdogan et al 2010). Similar to MapReduce/Hadoop applications, there are also considerable recent works on GPGPU applications to geospatial computing, including environmental modeling (Molna et al 2010), flow accumulation (Ortega and Rueda 2010),

[17] http://en.wikipedia.org/wiki/Message_Passing_Interface
[18] http://hadoop.apache.org/

drainage network computation (Qin and Zhan et al 2012), LIDAR data reduction (Oryspayev, in press) and raster analysis (Steinbach and Hemmerling, in press). 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 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's fast shared memory. 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. While 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 graphics such as iso-surface rendering and ray-tracing, quadtrees have been proposed to compress binary and gray scale 2D rasters (Samet 1985, Lin 1997, Chan and Chang 2004, Chung et al 2006) in computer graphics and image processing communities, respectively. However, we note that data structures and algorithms designed for compression are not necessarily suitable for 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[19] 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). The techniques are similar to indexing spatial-temporal vector geospatial data such as Historical R-Tree (Nascimento et al 1998), MV3R-Tree (Tao and Papadias 2001) and TPR-Tree (Saltenis 2000) from a methodological perspective. 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, such as the Region-of-Interest (ROIs) type queries that we have used for Web-based visual explorations (Zhang and You 2010a, Zhang and You 2010b).

Techniques such as linear quadtrees (Samet 1984) have been developed to externalized 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 2009) associates a set of species identifiers with linear quadtree nodes and uses the PostgreSQL LTREE module[20] to perform window queries by coordinating both the query client and the database server. A main-memory implementation has improved query performance by 2-

[19] http://docs.oracle.com/html/B10827_01/geor_intro.htm
[20] http://www.postgresql.org/docs/9.1/static/ltree.html

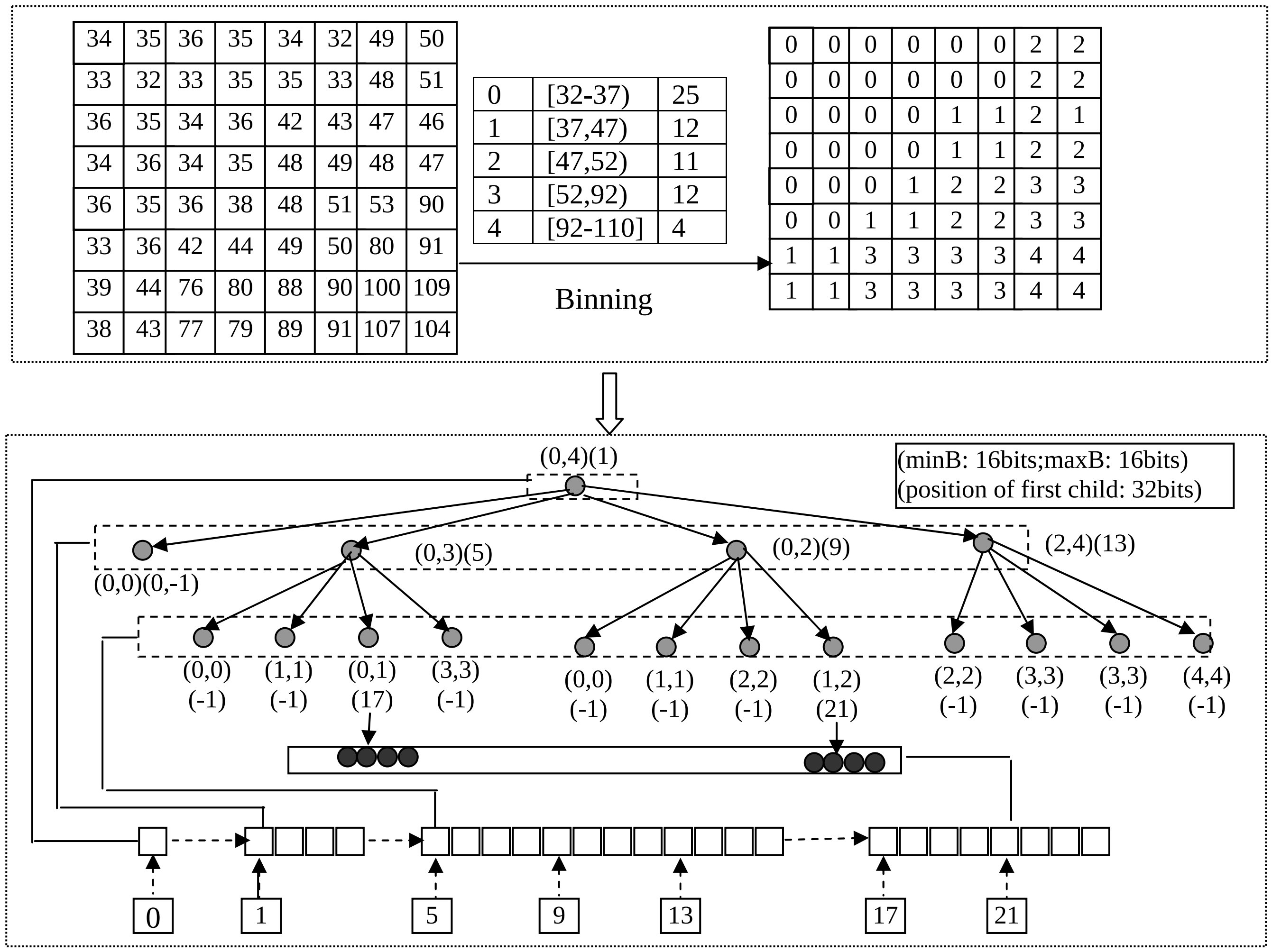
3 orders as reported in (Zhang 2012, also see online demo at[21]). A Binned Min-Max Quadtree (BMMQ-Tree) data structure that associates min/max statistics of raster cells of a quadrant to the corresponding quadtree node to speed up processing of certain types of queries in a Web environment has been developed (Zhang and You 2010a). BMMQ-Tree is a CPU main-memory data structure constructed through a recursive procedure. More recently, the BMMQ-Tree construction algorithm has been implemented on Nvidia GPUs using CUDA directly (Zhang et al 2010). In this paper, we will introduce the BMMQ-Tree data structure and its CUDA-based construction algorithm (i.e., the CUDA-Naïve implementation) in more details in Section 3 before we present our parallel primitives based implementation in Section 4.

3 BMMQ-Tree and its CUDA-based Construction

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 (min/max 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 nodes with an array index to point to the first child node. The layout of the BMMQ-Tree data structure is illustrated in Fig. 2. 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 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 traversal. As discussed in (Zhang et al 2010), the BMMQ-Tree structure is cache conscious since sibling nodes are consecutive in the data stream and is 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, are stored. More importantly, the quadtree data structure is GPU-friendly as the data stream of quadtree nodes can be easily held in a one dimensional array and transferred back and forth between CPU and GPU memories (as well as disks and CPU memories) without serialization.

The original implementation of the construction algorithm of the BMMQ-Tree tree using CUDA directly, i.e., the CUDA-Naïve implementation, is fairly complicated. The process first builds a min/max pyramid from the original raster data. For each cell of a raster at each level in the pyramid, the minB and maxB values are derived bottom-up. Starting from the top level of the pyramid, the algorithm then performs two Z-order (Morton 1966) based prefix sums (scans)[22] to compute the position of each of the nodes on the output data stream and fill the position of the first child node in the stream for each node. The first scan to fill the position of the first child node on the stream for each node is performed on a sequence of child numbers c_i where c_i is 0 if all minB and maxB values of the node are exactly the same (i.e., the quadrant is homogeneous and there is no need for further subdivision) and c_i is 4 otherwise. The second scan to compute the position of each of the node on the output data stream is performed on a sequence of values of p_i where p_i is 0 if c_i is 0 and p_i is 1 if c_i is not 0. An example of the parallel tree construction process is illustrated in Fig. 3.

[21] http://geoteci.engr.ccny.cuny.edu/geoteci/SPTestMap.html
[22] http://en.wikipedia.org/wiki/Prefix_sum



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	[92-110]	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 Layout of a BMMQ-Tree

While we refer to (Zhang et al 2010) for more details regarding to the CUDA-Naive implementation, we would like to note that quite some of our coding effort were spent on level-wise kernel launches. In order to fully utilize GPU threads, we had designed a sophisticated kernel launch schema. As shown in Fig. 4, at the finest level (K), for a raster dataset has a size of N*N (N=2^K), we have set the data grid dimension to P*P (P=2^L) and thread block dimension to T*T (T=2^U) and each thread was responsible for processing Q*Q raster cells (Q=2^{K-L-U}). During the bottom-up process to compute the min/max values of all pyramid elements, from level K to level L+U, the workload of each thread is reduced to 1/4 due to the aggregations of pyramid elements. At the level L+U, each thread will only process four elements of the level L+U+1 matrix in the pyramid. The sizes of the matrices between level L and level L+U are below the number of threads that are allocated to the previously defined kernel. To utilize the GPU device fully, the block size and thread size per block are reduced gradually by launching new kernels at each level. The top S levels of the tree were built in CPU as there was not enough parallelism to make full use of the GPU threads. As GPU cores are usually slower (with respect to clock rate) and less powerful (with respect to cache utilization and handling branching) than CPUs, it is more beneficial to process the top S levels in the CPU.

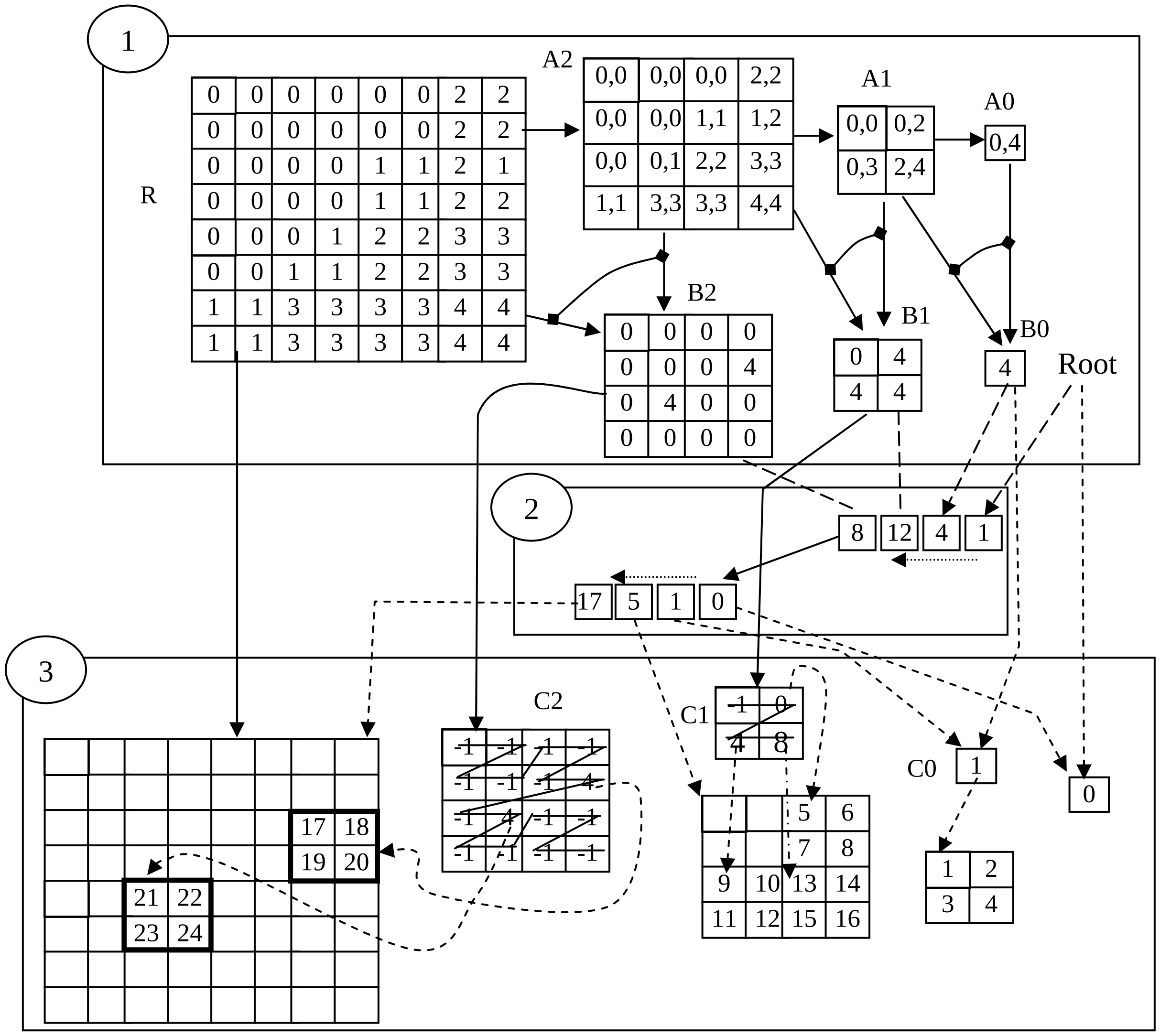


Fig. 3 Example of the parallel construction process of a BMMQ-Tree: min/max pairs and numbers of children are computed for all pyramid levels in step 1, positions of the first child nodes are computed by prefix-sums (scans) using the numbers of children array in step 2, and finally, positions of all the quadtree nodes in the output array are computed also through a prefix-sum (scan) process. The data fields are filled and the quadtree nodes are output in parallel to construct a BMMQ-Tree after the three steps.

Besides determining the schema to map GPGPU computing blocks and threads to raster cells, another key issue in the CUDA-Naive implementation was to perform Z-order based prefix-sums (scans) to compute proper values of the first child positions and the node positions as discussed earlier. Mixing Z-order based data accesses to pyramid matrixes and prefix-sums (scans) makes the code fairly complicated. While we were able to make the CUDA program work and gained signficant speedups using an Nvidia Quadro FX3700 GPU card over an Intel

CPU E5405 processor (only one core was used), it took us more than four months to complete the development. The resulting CUDA program has more than 1000 LOC (Lines of Code) and was not well structured. Another drawback of the CUDA-Naive implementation was that we were not able to use shared memory to speed up computation and data accesses from/to global memory. Despite we were aware that accesses to shared memory were about two orders faster than accesses to GPU global memory (Kirk and Hwu 2010), we were not able to identify data-reuse patterns and were not able to make use of GPU shared memory. As we shall see in the next section, by using the scan primitive where shared memory is utilized extensively and automatically, the tree construction performance can be improved significantly.

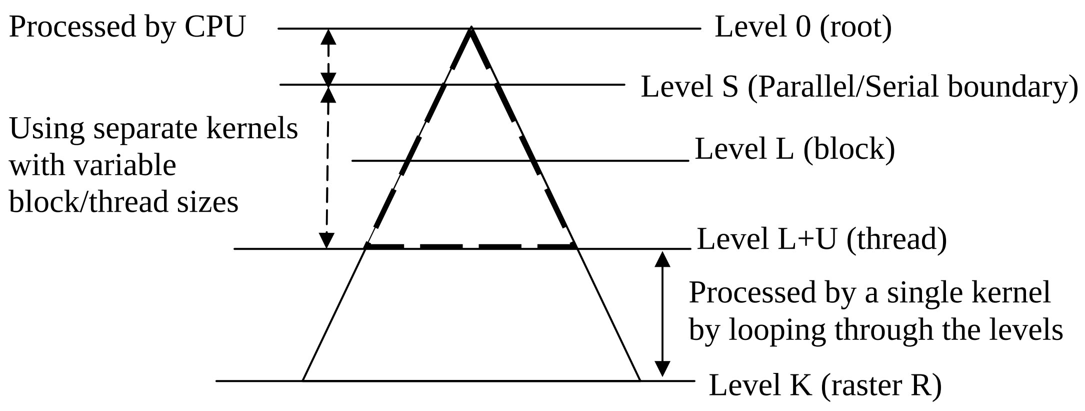
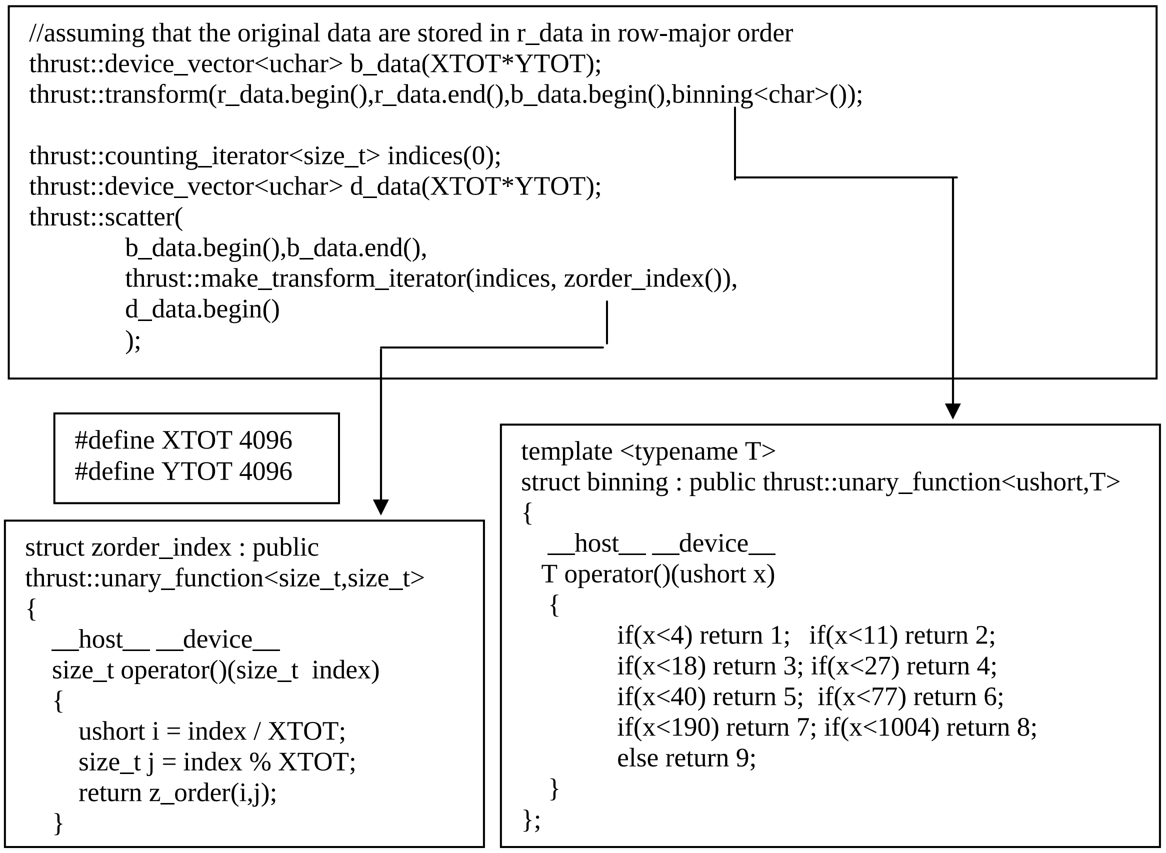

Processed by CPU
Level 0 (root)
Level S (Parallel/Serial boundary)
Using separate kernels with variable block/thread sizes
Level L (block)
Level L+U (thread)
Processed by a single kernel by looping through the levels
Level K (raster R)

Fig. 4 Kernel Launching Schema of the CUDA-Naive Implementation

4 Constructing BMMQ Tree using Parallel Primitives

Similar to the CPU construction of a BMMQ-tree, the construction process begins with a binning process as illustrated at the top part of Fig. 5. This can be easily implemented by using a transform primitive. For all the elements in the input vector (row-major ordered raster cells), a binning functor (C++ function object) is applied and the result is copied to the output vector. The binning process can be efficiently performed on GPUs. For example, our experiments have shown that binning a 16-bit 4096*4096 raster into a 8-bin raster takes about 1.32 milliseconds on an Nvidia Quadro 6000 card, i.e., 12.7 billion data elements (or 101 billion bins) per second . However, as parallel primitives that are currently available in most libraries (including Thrust) are mostly designed for 1D data, there is a semantic mismatch between using parallel primitives and processing multidimensional geospatial data. Fortunately, there are many well-established techniques to transform multidimensional data into space filling curves with known mathematical properties[23]. Among various space filing curve based orderings, Z-Order (Morton 1966) might be the simplest and easiest one. Our task here is thus to transform 2D raster cells into 1D vector elements based on the Z-Order. As shown in Fig. 5, this can be done by using the scatter primitive in conjunction a transform iterator and a functor. As introduced in Section 2.2, a scatter primitive copies elements from a source range into an output vector according to a map.

[23] http://en.wikipedia.org/wiki/Space-filling_curve



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

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()
            );
```

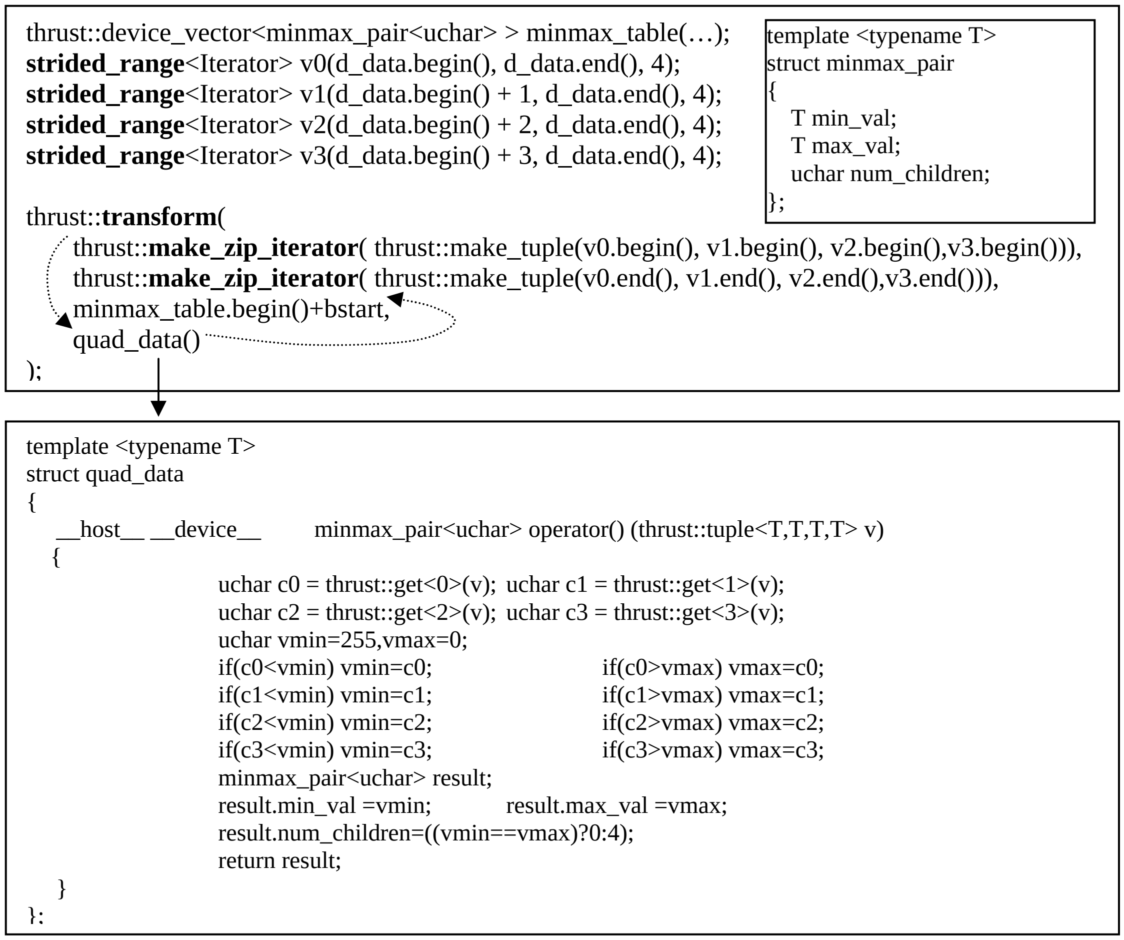
```
#define XTOT 4096
#define YTOT 4096
```

```
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 % XTOT;
      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. 5 Code Segment to Illustrate Z-order Based Transformation Using the *Scatter* Primitive

While it is easy to observe that the source vector is b_data (the first and the second parameters) and the destination range is d_data (the fourth parameter) in calling the *scatter* primitive at middle part of Fig. 5, we would like to provide more details on how the map vector (the third parameter) is built. In fact, the map vector is provided as a transform iterator that transforms row-major order (represented by a *counting_iterator* variable indices) to Z-order on-the-fly by calling operator() of the user-defined zorder_index functor. We have used the lookup approach introduced in (Raman and Wise 2008) to implement the zorder_index functor for efficiency purpose. When the *scatter* primitive is executed, for each element in b_data, in parallel, a Z-order value is computed based on its index position in b_data and its value is copied to the index position of the Z-order value in the d_data vector. Obviously there is a one-to-one correspondence between the elements in b_data and d_data but elements in b_data follow a row-major order and elements in d_data follow a Z-order. Subsequently d_data is used as the new input data for further processing. We note that it is generally non-trivial to transform orders of data elements on GPGPUs efficiently as efficiency can be significantly decreased if GPU memories are not accessed in a coalesced manner, i.e., consecutive threads need to access consecutive memory addresses. Fortunately, this is taken care of by the Thrust library that implements the *scatter* primitive. A variety of sophisticated optimization techniques, such as kernel configuration and using shared memory for intermediate results, are applied in the Thrust library but the optimizations are transparent to application developers. These types of “automatic optimization" can signficant reduce development complexities and improve performance at the same time.



```
thrust::device_vector<minmax_pair<uchar> > minmax_table(…);
strided_range<Iterator> v0(d_data.begin(), d_data.end(), 4);
strided_range<Iterator> v1(d_data.begin() + 1, d_data.end(), 4);
strided_range<Iterator> v2(d_data.begin() + 2, d_data.end(), 4);
strided_range<Iterator> v3(d_data.begin() + 3, d_data.end(), 4);

thrust::transform(
    thrust::make_zip_iterator( thrust::make_tuple(v0.begin(), v1.begin(), v2.begin(),v3.begin())),
    thrust::make_zip_iterator( thrust::make_tuple(v0.end(), v1.end(), v2.end(),v3.end())),
    minmax_table.begin()+bstart,
    quad_data()
);
```

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

```
template <typename T>
struct quad_data
{
   __host__ __device__      minmax_pair<uchar> operator() (thrust::tuple<T,T,T,T> v)
   {
                   uchar c0 = thrust::get<0>(v);  uchar c1 = thrust::get<1>(v);
                   uchar c2 = thrust::get<2>(v);  uchar c3 = thrust::get<3>(v);
                   uchar vmin=255,vmax=0;
                   if(c0<vmin) vmin=c0;             if(c0>vmax) vmax=c0;
                   if(c1<vmin) vmin=c1;             if(c1>vmax) vmax=c1;
                   if(c2<vmin) vmin=c2;             if(c2>vmax) vmax=c2;
                   if(c3<vmin) vmin=c3;             if(c3>vmax) vmax=c3;
                   minmax_pair<uchar> result;
                   result.min_val =vmin;         result.max_val =vmax;
                   result.num_children=((vmin==vmax)?0:4);
                   return result;
   }
};
```

Fig. 6 Code Segment to Illustrate Generating Bottom-Level Min/Max Table from a Binned Raster using a *Transform* Primitive

The next step is to derive the bottom level min/max matrix of the raster pyramid from the raw data. We first divide the 1D raster cell vector (now in Z-order) into four vectors with the ith vector takes 0*4+i, 1*4+i, 2*4+i …, S*4+i elements in the original vector where S=N*N/4. This can be done in Thrust using a class called strided_range. Subsequently for raster cells from the four vectors can be used to make a tuple and the constructor of the quad_data functor can be applied as shown in Fig. 6. Note that the make_tuple function usually goes together with the make_zip_iterator function to construct a virtual vector of structures which has quite some performance advantages (Bell and Hoberock 2011). The output of the transform primitive is written to the min/max table vector (minmax_table) starting at bstart=$1+4+4^2+\ldots+4^{K-2}=(4^{K-1}-1)/3$ where K is the height of the raster pyramid. Note that the minmax_table vector is used to store the min-max pairs for all the (K-1) levels and the length of the vector is blen=$1+4+4^2+\ldots+4^{K-1} = (4^K-1)/3$. Concatenating all the min-max pair vectors into a big one not only make it easier than maintaining K-1 separate vectors but also make it possible to construct all tree nodes using a single parallel primitive as we shall detail shortly. For the four raster cells in a quadrant, the operator() of the quad_data functor returns a min-max pair of the cell values by using a transform primitive. During this process, the number of children at the quadrant is also counted in the quad_data functor to form a minmax_piar structure. If all the four cells under a quadrant in the original raster have the same value, then the quadrant should have 0 children, otherwise it should have 4 children. Since we have already had the minimum and maximum values of the quadrant, the criteria to determine the number of children can be simplified as the following simple statement: result.num_children=((vmin==vmax)?0:4). Note that the quad_data functor in the *transform* primitive takes an element of the input vector, i.e., the virtual vector constructed by the combination of the make_tuple and make_zip_iterator functions, and compute a minmax_pair structure to be written to the output vector, i.e., minmax_table. The relationships among the input vector, the functor and the output vector are marked with arrows in the middle part of Fig. 6. The same illustration has also been provided in Fig. 7 through Fig. 9.

After the min-max table for the bottom level of the raster pyramid is constructed, we can follow pretty much the same procedure to construct the min-max tables of the rest levels of the raster pyramid in a bottom-up manner (Fig. 7). Since we concatenate all K-1 levels of the min-max pairs in a vector (minmax_table), we need to compute the starting positions and sizes of the min-max tables at all levels. At each pyramid level (k), we can compute the starting position as c_start=$1+4+4^2+\ldots+4^k=((4^{k+1}-1)/3$ and raster size as c_size= $2^{k+1}*2^{k+1}=4^{k+1}$. Four min-max pairs at level k are then aggregated and output to the minmax_table vector at level k-1 starting at p_start=$(4^k-1)/3$. The logics of computing the min-max pairs at each level of the pyramid are very similar to computing the min-max pairs from the raw data vector shown in Fig. 6. The only required change is the function to be applied to each cell at each pyramid level. The functor to be applied (quad_pyra) is very similar to the quad_data functor in Fig. 6 with respect to computing the min/max values from the four minmax_pair structures in a quadrant. We note that the strided_range class is again used to split the minmax_pair structures in the original vector (minmax_table) into four strided vectors. The four strided vectors are subsequently used to make tuples in order to use the zip iterators in the *transform* primitive (middle part of Fig. 7).

While it is straightforward to construct the min-max table level-wise and bottom up, a subtle issue is how to determine whether the quadtree nodes corresponding to the minmax_pair structures in the min-max table at the all pyramid levels should be kept or not. Conceptually only a highest level quadtree node is kept to represent a homogenous quadrant while all the nodes under it should be pruned. This can be easy done in the CUDA-naïve implementation by setting a flag in all the child nodes under a parent node if the parent node is determined to represent a homogenous quadrant. Unfortunately, Thrust parallel primitives usually do not allow change elements in input vectors for both efficiency and safety considerations. Our solution, as shown in Fig. 8, is to extract the number of child nodes of parent nodes into a separate vector (stored in tmp_numchild in Fig. 8) and use the vector to determine whether the respective child nodes should be kept in the resulting quadtree based on the following fact. If a parent node represents a non-homogeneous quadrant (i.e., number of children is 4) but a child node represents a homogeneous region (i.e., number of children is 0), then the child node is a leaf node in the resulting quadtree and should be kept. On the other hand, if both the child node and its parent node represent homogeneous regions then the child node should not be a node in the resulting quadtree. Our implementation reuses the child node number field as an indication flag. For the minmax_pair structures in the min-max table corresponding to the quadtree leaf nodes, we set their child node numbers to a large value (e.g., 255) in the update_numchild functor in the second transform primitive in Fig. 8.

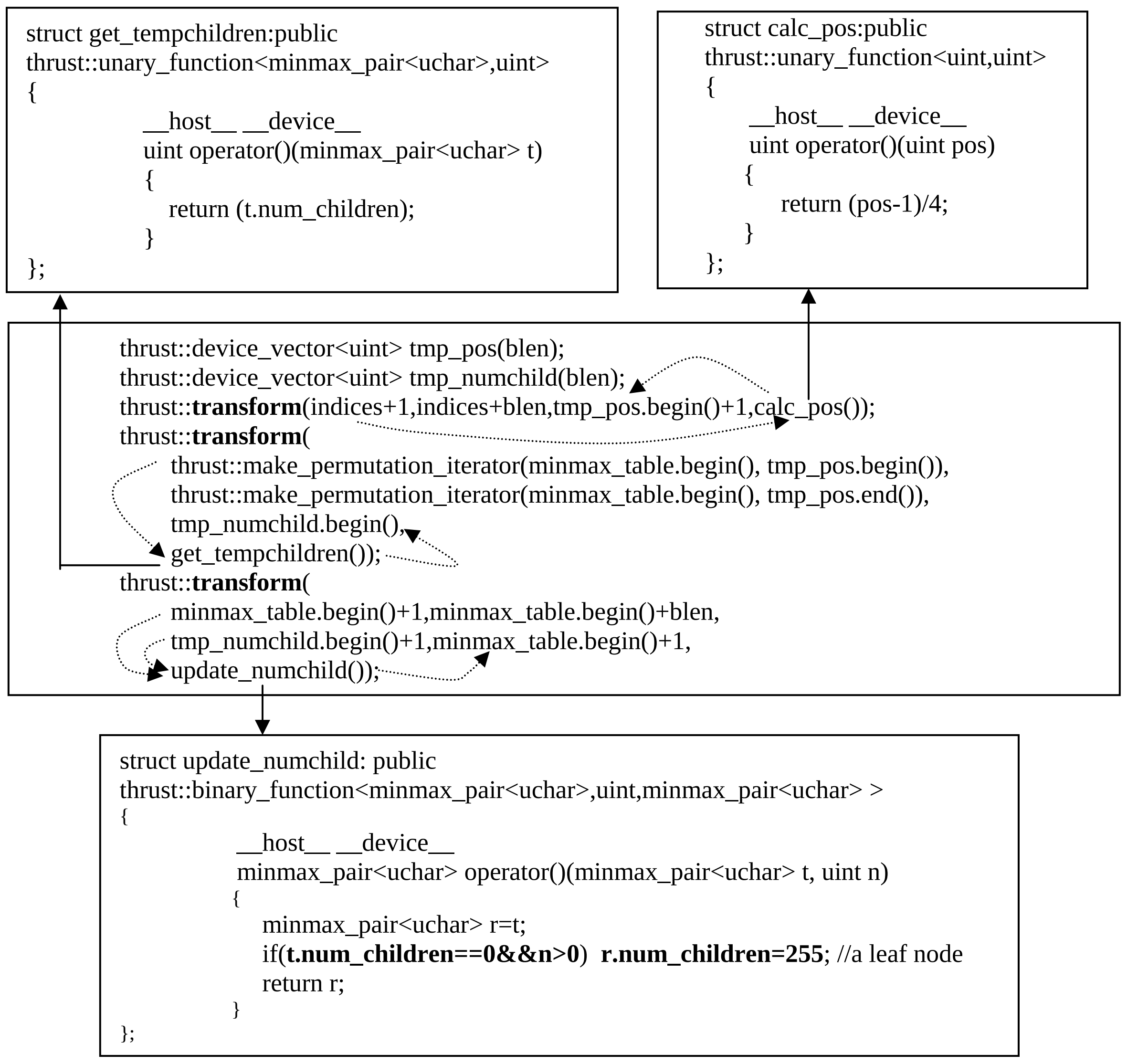
```
for(int k=M-2;k>=0;k--)
 {
       int c_start=(int)((std::pow(4.0f,k+1)-1)/3);
       int p_start=(int)((std::pow(4.0f,k)-1)/3);
       int c_size=(int)(std::pow(4.0f,k+1));
       strided_range<MMIterator> u0(minmax_table.begin()+c_start+0,
               minmax_table.begin()+c_start+c_size, 4);
       strided_range<MMIterator> u1(minmax_table.begin()+c_start+1,
               minmax_table.begin()+c_start+c_size, 4);
       strided_range<MMIterator> u2(minmax_table.begin()+c_start+2,
               minmax_table.begin()+c_start+c_size, 4);
       strided_range<MMIterator> u3(minmax_table.begin()+c_start+3,
               minmax_table.begin()+c_start+c_size, 4);
       thrust::transform(
               thrust::make_zip_iterator(thrust::make_tuple(u0.begin(), u1.begin(), u2.begin(),u3.begin())),
               thrust::make_zip_iterator(thrust::make_tuple(u0.end(), u1.end(), u2.end(),u3.end())),
               minmax_table.begin()+p_start,
               quad_pyra());
```

```
template <typename T>
struct quad_pyra
{
    __host__ __device__
    minmax_pair<uchar> operator()(thrust::tuple<minmax_pair<T>,minmax_pair<T>,
                minmax_pair<T>,minmax_pair<T> > v)
   {
       minmax_pair<uchar> c0 = thrust::get<0>(v); minmax_pair<uchar> c1 = thrust::get<1>(v);
       minmax_pair<uchar> c2 = thrust::get<2>(v); minmax_pair<uchar> c3 = thrust::get<3>(v);

       T vmin=255,vmax=0;
       if(c0.min_val<vmin) vmin=c0.min_val;      if(c0.max_val>vmax) vmax=c0.max_val;
       if(c1.min_val<vmin) vmin=c1.min_val;      if(c1.max_val>vmax) vmax=c1.max_val;
       if(c2.min_val<vmin) vmin=c2.min_val;      if(c2.max_val>vmax) vmax=c2.max_val;
       if(c3.min_val<vmin) vmin=c3.min_val;      if(c3.max_val>vmax) vmax=c3.max_val;

       minmax_pair<T > result;
       result.min_val =vmin;      result.max_val =vmax;
       result.num_children=((vmin==vmax)?0:4);
       return result;
   }
};
```

Fig. 7 Code Segment to Illustrate Generating Min/Max Table at the All Pyramid Levels Bottom-Up using a *Transform* Primitive



```
struct get_tempchildren:public
thrust::unary_function<minmax_pair<uchar>,uint>
{
                __host__ __device__
                uint operator()(minmax_pair<uchar> t)
                {
                   return (t.num_children);
                }
};
```

```
struct calc_pos:public
thrust::unary_function<uint,uint>
{
      __host__ __device__
      uint operator()(uint pos)
      {
          return (pos-1)/4;
      }
};
```

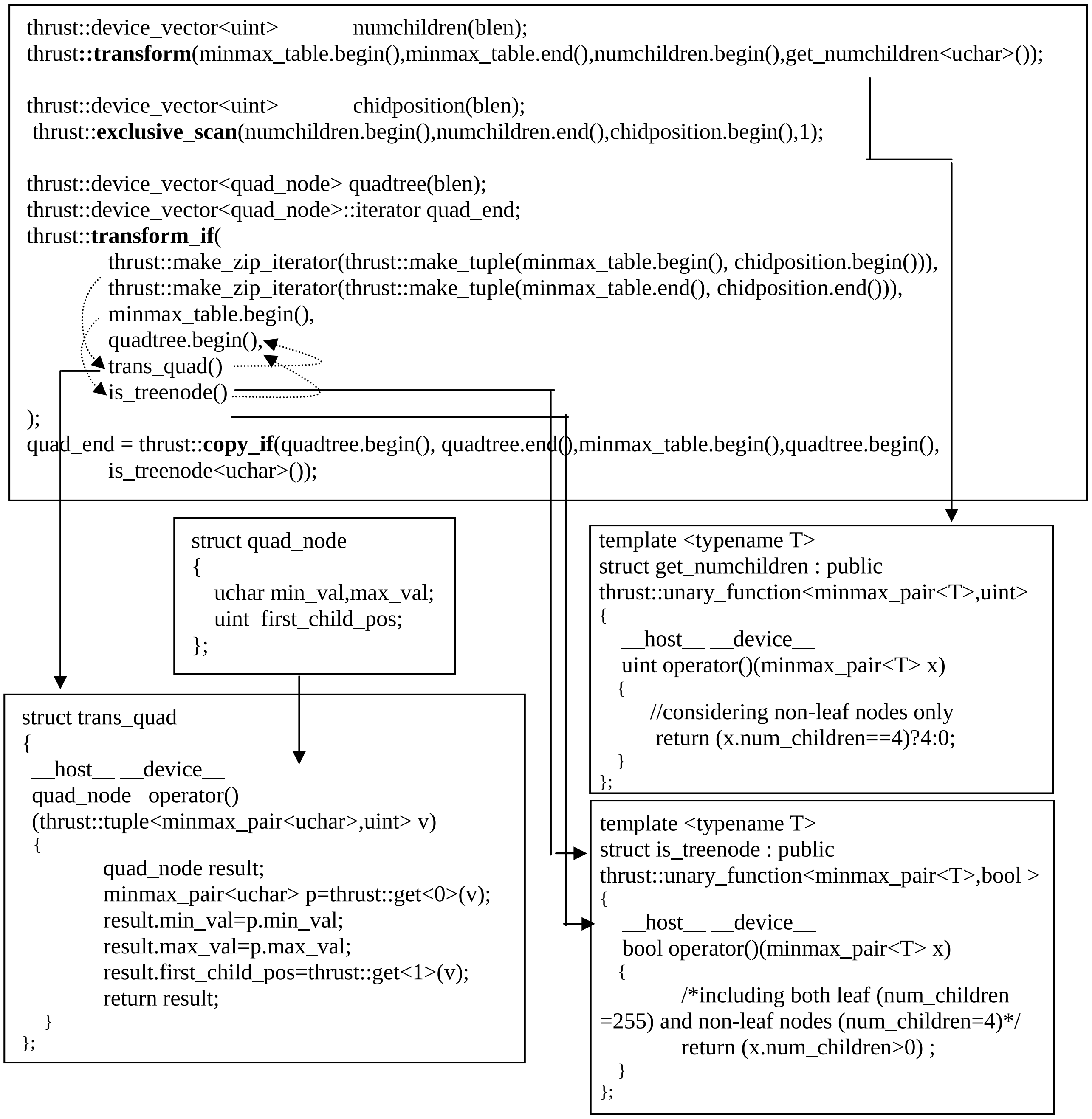
```
thrust::device_vector<uint> tmp_pos(blen);
thrust::device_vector<uint> tmp_numchild(blen);
thrust::transform(indices+1,indices+blen,tmp_pos.begin()+1,calc_pos());
thrust::transform(
       thrust::make_permutation_iterator(minmax_table.begin(), tmp_pos.begin()),
       thrust::make_permutation_iterator(minmax_table.begin(), tmp_pos.end()),
       tmp_numchild.begin(),
       get_tempchildren());
thrust::transform(
       minmax_table.begin()+1,minmax_table.begin()+blen,
       tmp_numchild.begin()+1,minmax_table.begin()+1,
       update_numchild());
```

```
struct update_numchild: public
thrust::binary_function<minmax_pair<uchar>,uint,minmax_pair<uchar> >
{
                __host__ __device__
                minmax_pair<uchar> operator()(minmax_pair<uchar> t, uint n)
                {
                    minmax_pair<uchar> r=t;
                    if(t.num_children==0&&n>0)  r.num_children=255; //a leaf node
                    return r;
                }
};
```

Fig. 8 Code Segment to Illustrate Identifying Leaf Nodes

Note that this version of the *transform* primitive takes two input vectors (minmax_table and tmp_numchild, respectively) and the update_numchild functor is binary which produces an output vector (minmax_table) based on the two input vector. Since the output vector here is the same as the first input vector, the transform is done in place. The alert reader might observe that, the tmp_numchild, which stores the numbers of child nodes of parent nodes, has the same length as that of minmax_table. As one parent node has four child nodes in quadtrees, the child node numbers must have been replicated in order to make it possible to compare the child node numbers between all parent-child pairs in parallel using the transform primitive. Indeed, this is the trickiest part of the primitives based implementation which is done by integrating two *transform* primitives, one further makes use of a *counting_iterator* and the other further uses a *permutation_iterator*, as shown at the top of Fig. 8. The first *transform* primitive takes a vector of sequential numbers (to be dynamically generated by the *counting_iterator* variable indices) which represents all child node positions and the output vector is tmp_pos which represents the respective parent node positions. The map between the input and output is computed by the calc_pos functor which is simply parent_pos=(child_pos-1)/4. After the parent node positions are calculated, the second *transform* primitive actually copies the numbers of children in the parent nodes to these of the child nodes based on the positions. The role of the *permutation_iterator* is similar to the *Scatter* primitive which generates an output vector based on an input vector and a map on the fly while the *permutation_iterator* is advanced (in parallel). In our implementation, the value of the number of children field of the minmax_pair structures in the min/max table vector has three possible values: 255 indicates a leaf node, 4 indicates a non-leaf nodes and 0 indicates a non-tree node. Only the numbers of child nodes of non-leaf nodes are used to compute child node positions but both the leaf and non-leaf nodes are used to construct a quadtree.

After the min-max pair vector is constructed in parallel for all levels of the raster pyramid, we proceed to the stage of assembling a BMMQ tree where we need to fill the minB, maxB and the first child node position fields for all tree nodes. Similar to what we have done previously (Zhang et al 2010), a prefix-sum (scan) on the numbers of children of all non-leaf tree nodes will serve the purpose of computing the positions of the first child nodes. The prefix-sum can be easily done in a single call to the *exclusive_scan* primitive after calling the *transform* primitive to compute the numbers of children for all tree nodes as shown at the top of Fig. 9. We also refer to Fig. 1 for the illustration of the implementation of the scan primitive. The last step to construct a BMMQ tree is to actually fill the min/max values and the first child node positions in the array of the tree nodes by using a transform primitive. The *transform_if* primitive takes two input vectors, i.e., the combination of minmax_table and chidposition through make_zip_iterator and make_tuple and the minmax_table alone, and two unary functors, i.e., quad_node and is_treenode. The quad_node binary functor takes a tuple of a minmax_pair structure in the minmax_table and a value in the chidposition vector and outputs the resulting quad_node structure to the output vector (i.e., *quadtree*) if the is_treenode functor is evaluated to true by using the corresponding element in the minmax_table vector (i.e., a min-max pair) as the input. Since we only want to keep valid tree nodes in the final quadtree output vector, the tree node pruning logic is implemented in the is_treenode functor by examining whether the number of children that is associated with the corresponding min-max pair is greater than zero which covers both the case of non-leaf nodes (number of children is 4) and the case of leaf node (number of children is purposely set to 255). Although the elements in the quadtree output vector are not updated if the is_treenode is evaluated to false at the position (which is more efficient), they still occupy spaces in the quadtree vector whose values are set when the vector is initialized. The quadtree vector can be compacted by using an in-place *copy_if* primitive on the quadtree vector which reuses the is_treenode functor as its unary predicate to evaluate the corresponding element in the minmax_table vector.



```
thrust::device_vector<uint>            numchildren(blen);
thrust::transform(minmax_table.begin(),minmax_table.end(),numchildren.begin(),get_numchildren<uchar>());

thrust::device_vector<uint>            chidposition(blen);
 thrust::exclusive_scan(numchildren.begin(),numchildren.end(),chidposition.begin(),1);

thrust::device_vector<quad_node> quadtree(blen);
thrust::device_vector<quad_node>::iterator quad_end;
thrust::transform_if(
          thrust::make_zip_iterator(thrust::make_tuple(minmax_table.begin(), chidposition.begin())),
          thrust::make_zip_iterator(thrust::make_tuple(minmax_table.end(), chidposition.end())),
          minmax_table.begin(),
          quadtree.begin(),
          trans_quad()
          is_treenode()
);
quad_end = thrust::copy_if(quadtree.begin(), quadtree.end(),minmax_table.begin(),quadtree.begin(),
          is_treenode<uchar>());
```

```
struct quad_node
{
   uchar min_val,max_val;
   uint  first_child_pos;
};
```

```
template <typename T>
struct get_numchildren : public
thrust::unary_function<minmax_pair<T>,uint>
{
   __host__ __device__
   uint operator()(minmax_pair<T> x)
   {
        //considering non-leaf nodes only
         return (x.num_children==4)?4:0;
   }
};
```

```
struct trans_quad
{
 __host__ __device__
 quad_node   operator()
 (thrust::tuple<minmax_pair<uchar>,uint> v)
 {
          quad_node result;
          minmax_pair<uchar> p=thrust::get<0>(v);
          result.min_val=p.min_val;
          result.max_val=p.max_val;
          result.first_child_pos=thrust::get<1>(v);
          return result;
   }
};
```

```
template <typename T>
struct is_treenode : public
thrust::unary_function<minmax_pair<T>,bool >
{
   __host__ __device__
   bool operator()(minmax_pair<T> x)
   {
          /*including both leaf (num_children
=255) and non-leaf nodes (num_children=4)*/
          return (x.num_children>0) ;
   }
};
```

Fig. 9 Code Segment to Illustrate the Process of BMMQ-Tree Construction by Integrating Transform, Scan and Copy Primitives

5 Evaluations and Discussions

We evaluate the proposed primitive-based BMMQ-Tree construction approach using two measurements: coding complexity and code efficiency by comparing them with the CUDA-naive implementation (Zhang et al 2010). We use the same global 30-arcseconds January Precipitation dataset from WolrdClim website[24]. Since dataset is divided into 4096*4096 tiles in CUDA-naïve implementation based experiments, we apply the same tiling schema in this study. We note that while the time complexity of the tree construction algorithm varies with the tile sizes and the output quadtree sizes depends on the input raster data, for the rasters that have a same dimension, the runtimes are largely input independent. The major workloads to construct the min/max table, computing first child node positions and filling quadtree nodes remain the same for any input raster with a same raster grid dimension. The differences for the last copy_if primitive are relatively insignificant. As such, although we have experimented on multiple tiles and multiple rasters, we will only report our experiment results on a 4096*4096 tile from the global 30-arcseconds January Precipitation dataset. Additional experiment results are available in our project website[25]. Since 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 an Nvidia Quadro 6000 GPU card[26] with 448 cores and 6 gigabytes global memory.

With respect to coding complexity, while the measurements and observations might be subjective, we would conclude that the coding complexity is significantly reduced when the primitive based implementation is compared with the previous CUDA-naive implementation. The CUDA-naive implementation has about 1000 LOC (lines of code). Many of them are fairly long lines especially for those involve array subscript calculations in Z-order transformations. In contrast, excluding the data preparation part, the main-body of the primitive based implementation is only about 50 lines, including both calls to parallel primitives and control flows. An additional 200 lines are needed to implement the functors that work with the primitives (as shown in Figs.5-9). However, the functors are fairly simple and the implementations are straightforward. While it took us sometime to learn the Thrust library, writing functors is much easier than embedding complex logics in CUDA statements. More importantly, the code is better structured, logically clearer and subsequently easier to debug and maintain. Furthermore, since we do not program GPU directly, understanding GPU hardware details is optional rather than mandatory. In general, we believe that the parallel primitives based implementation is clearly the winner with respect to coding complexity. Although the learning curve on the parallel primitives still exists, the curve is not as steep as that of CUDA.

In terms of code efficiency, while it is well-known that calling primitive libraries inevitably incur signficant overheads and is generally considered to be less efficient than native implementations, our experiments have shown differently. It takes about 130 milliseconds to construct a BMMQ-Tree for a 4096*4096 raster tile using the CUDA-naive implementation. The runtime dropped to about 13.5 milliseconds using the parallel primitives based implementation, i.e., approximately a 10X speedup is observed. Looking into the breakdowns of the runtimes, we find that the CUDA-naïve implementation actually is a little faster in the last step, i.e., assembling min/max values and first child node positions into quadtree nodes that corresponds to the last two primitives (transform and copy_if) in Fig. 9. The result is not surprising. However, the primitives based implementation is much faster than the CUDA-naive implementation with respect to deriving min/max tables and computing first child node positions. Although it additionally took about 3.25 milliseconds to perform the Z-order transformation, it only took 1.62 milliseconds to compute the bottom level of the min/max table from the corresponding raw

[24] http://biogeo.ucdavis.edu/data/climate/worldclim/1_4/grid/cur/prec_30s_bil.zip
[25]http://134.74.112.65/primquad/primquad.htm
[26] http://www.nvidia.com/object/product-quadro-6000-us.html

raster cells. In contrast, it took 51 milliseconds for the CUDA-naive implementation in this step. Similarly, computing all the other levels of the min/max table took only 0.523 milliseconds in the primitives based implementation while it was 27 milliseconds in the CUDA-naive implementation. With respect to computing the first child node positions, the runtimes are 2.05 milliseconds for the primitives based implementation and 42 milliseconds for the CUDA based implementation, respectively.

Where does this 10X speedup come from? A careful examination of both implementations reveals that two factors play a key role. First, as discussed before, the CUDA-naive implementation accesses GPU global memory in a row-major order when calculating the min/max table in a way similar to what we normally program on CPUs. This is quite typical for CUDA beginners and non-experts. However, the Single Instruction Multiple Thread (SIMD) parallel model adopted in CUDA has a quite different optimal memory access requirement. Different from CPUs where multi-levels of caches are utilized to automatically capture spatial and temporal access patterns and optimize data movements for a single processor, when GPU global memories are not accessed in a coalesced manner that are friendly to warps of threads, the signficant speed gap between GPU global memory and registers can be two orders or more and the performance loose can be significant (Kirk and Hwu 2010). Since consecutive threads only access two consecutive raster cells in the CUDA-naive implementation, the memory accesses are not coalesced and the access pattern is far from optimal. In contrast, in the parallel primitive based implementation, after transforming a 2D raster into a 1D vector, the majority of the operations performed by the respective primitives, such as transform and copy, are naturally memory access friendly. Additional optimizations have been provided by the Thrust library to further improve memory access efficiency on 1D vectors. The second factor is automatically using shared memory in Thrust primitives. Take the exclusive scan for example, Thrust first determines a good kernel configuration schema, such as number for computing blocks and the number of threads per block. Within each computing block, after data are collaboratively copied from global memory to per-block shared memory (as shown in Fig. 1), the data are summed up using shared memory as the scratch space before the results are written back collaboratively to global memory. The way Thrust does the scan is much more efficient than the embedded scans in the CUDA-naive implementation where all computations are performed on global memory without using the fast shared memory. Since the access latency to shared memory is comparable to that of registers and is about two orders faster than accesses to global memory (Kirk and Hwu 2010), the primitives based calculation of first child node positions has achieved a 20X speedup (42 milliseconds versus 2.05 milliseconds).

We note that it is certainly possible to incorporate both the techniques (coalesced memory accesses and using shared memories) that are utilized by the Thrust primitives into the CUDA-naive implementation. It is quite likely that an optimized CUDA based implementation can achieve a signficant speedup over the primitives based implementation after performance profiling and applying optimizations. However, we argue that such optimization and performance tuning can be time consuming even for CUDA experts. Given that there are no existing guidelines on optimizing memory accesses and computation for multidimensional geospatial data, such optimization techniques can be ad-hoc, tricky and highly dependent on developers' understanding of GPU hardware and CUDA programming model. As such, we argue that using highly optimized parallel primitives can be more beneficial to application developers in terms of coding complexity and code efficiency. A disadvantage of the primitives based approach is that application developers are responsible to map between geospatial problems onto the available parallel primitives. As discussed at the beginning of Section 4, there is a semantic gap between the multidimensional nature of geospatial data and the 1D vectors that are supported by most parallel libraries. While some parallel libraries do support multidimensional image data, such as Nvidia Performance Primitives (NPP)[27] , they are mostly designed to speed up local or focal operations on RGB values of image pixels while largely leaving zonal and global operations untouched. It is unclear how such existing libraries can help indexing and query processing of geospatial data. As such, we believe that it is important to learn from both the generic parallel primitives on 1D data and the newly emerged parallel primitives on multidimensional image data and develop geospatial specific parallel primitives to facilitate zonal and global operations based geospatial computing, including indexing and query processing of large-scale rasters. Such geospatial specific parallel primitives are likely to capture commonly used geospatial computing patterns better and achieve better performance at the same time. A community effort is required to build such high-performance geospatial computing primitives which will in turn benefit the geospatial computing community once such primitives are developed. A cyberinfrastructure approach is further required to understand community needs, seek consensuses and prioritize development efforts as well as disseminating research and development results to the community. These are left for our future work.

Conclusion and Future Works

Despite the emerging wide interests in applying GPGPU technologies to solve large geospatial problems as GPU hardware resources become increasingly available, the complexities of the unique GPGPU hardware architectures and the steep learning curve of GPGPU technologies have imposed signficant challenges on the geospatial computing community. One of such challenges is how to balance between coding complexity and code efficiency to use such parallel hardware resources effectively. Based on previous research on constructing a quadtree indexing structure for visually exploring large-scale raster data on both CPUs and GPUs, in this study, we have advocated for a primitives based approach. Our experiment results have shown that the parallel primitives based approach can significantly reduce coding complexity and improve code efficiency at the same time.

The parallel primitives based approach requires mapping between multidimensional geospatial data and 1D vectors that can be efficiently supported by existing parallel libraries. While the mapping is nontrivial, it provides a good opportunity to force geospatial computing researchers and developers to start to think about and understand the inherent parallelisms in geospatial data and geospatial computing. Such understanding is crucial in developing more efficient and geospatial-specific parallel primitives to bridge between conceptual deigns of geospatial computing models, software developments and hardware parallel executions.

The paper has taken a case study approach to introduce a software development framework that can effectively lower the barrier of harnessing GPGPU parallel computing power for geospatial computing. There are plenty of rooms left for future works. First of all, we would like to extend the quadtree based indexing for query processing on GPGPUs, including both individual and batched queries (e.g., location/window queries and spatial joins). Second, while we focus on raster data in this study, it is natural to extend the idea to point and polygonal data, including both indexing and query processing. Third, although we have been using a single GPU device for our data structure and algorithm development in a personal computing environment,

[27] http://developer.nvidia.com/npp

we plan to extend the approach to a cluster computing environment using grid/cloud computing resources to further test the scalability of the proposed approach. Finally, as discussed earlier, we have strong interests 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.

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).
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., 2011. 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. He, B. S., Lu,M. , et al., 2009. Relational Query Coprocessing on Graphics Processors. ACM Transactions on Database Systems 34(4).

18. Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann.
19. Hoel, E.G. and Samet, H., 2003. Data-parallel polygonization. Parallel Computing 29(10) 1381–1401
20. 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.
21. Hou, Q., Sun, X., et al., 2011. Memory-Scalable GPU Spatial Hierarchy Construction. IEEE Transactions on Visualization and Computer Graphics 17(4): 466-474.
22. Hwu, W.-M. W (eds.), 2011a. GPU Computing Gems: Emerald Edition. Morgan Kaufmann
23. Hwu, W.-M. W (eds.), 2011b. GPU Computing Gems: Jade Edition. Morgan Kaufmann
24. Kamel, I. and Faloutsos, C., 1992. Parallel r-trees. Proceedings of the ACM SIGMOD International conference on Management of data (SIGMOD'92) 195–204
25. Kirk, D. B. and Hwu, W.-M., 2010. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann.
26. Lin, T. W., 1997. Compressed quadtree representations for storing similar images. Image and Vision Computing 15(11), 833-843.
27. 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).
28. Manolopoulos Y., Nardelli, Y., E., et al., 2001. A generalized comparison of linear representations of thematic layers. Data & Knowledge Engineering 37(1): 1-23.
29. 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.
30. Molnár, F., T. Szakály, et al., 2010. Air pollution modelling using a Graphics Processing Unit with CUDA. Computer Physics Communications 181(1): 105-112.
31. Morton, G.M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. IBM Technical report.
32. Nascimento, M. A. and Silva, J. R. O., 1998. Towards historical R-trees. Proceedings of the 1998 ACM symposium on Applied Computing, 235 – 240
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., in press. LiDAR data reduction using vertex decimation and processing with GPGPU and multicore CPU technology. Computers and Geosciences.
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. Raman, R. and Wise, D.S., 2008. Converting to and from Dilated Integers. IEEE Transactions on Computers, 57(4) 567-573.
37. Saltenis, S., Jensen, C. S., et al., 2000. Indexing the positions of continuously moving objects. Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD'00).
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., in press. Accelerating batch processing of spatial raster analysis using GPU. Computers and Geosciences.
43. Tao, Y. and Papadias, D., 2001. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01).
44. Theobald, D. M., 2005. GIS Concepts and ArcGIS Methods, 2nd Ed., Conservation Planning Technologies, Inc.
45. 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).
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. Yan, L., W. Kaichao, et al., 2010. A MapReduce approach to G_i*(d) spatial statistic. Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems (HPDGIS'10).
53. Yang, C. W., Goodchild, M. A, et al., Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing. International Journal of Digital Earth, 4(4): 305-329.
54. Yang, C. W., Raskin, R. and Goodchild, M. A., 2009. Geospatial cyberinfrastructure: Past, present and future. Computers, Environment and Urban Systems, 34(4):264–277.
55. 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.
56. 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).
57. 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).

58. 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).
59. 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).
60. 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)
61. 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)
62. Zhang, J,, Gertz, M., Gruenwald, 2009. 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)
63. Zhang, S., J. Han, et al., 2009. SJMR: Parallelizing spatial join with MapReduce on clusters. Proceedings of the IEEE International Conference on Cluster Computing Workshops (CLUSTER '09).
64. Zhou, X., Abel, D. J., and Truffet, D., 1998. Data partitioning for parallel spatial join processing. GeoInformatica 2(2) 175–204
65. Zhou, K., Hou, Q., et al., 2008. Real-Time KD-Tree Construction on Graphics Hardware. ACM Transactions on Graphics 27(5).
66. Zhou, K., Gong, M., et al., 2011. Data-Parallel Octrees for Surface Reconstruction. IEEE Transactions on Visualization and Computer Graphics 17(5): 669-681.