High-Performance Zonal Histogramming on Large-Scale Geospatial Rasters Using GPUs and GPU-Accelerated Clusters

Jianting Zhang
Department of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Dali Wang
Environmental Science Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
wangd@ornl.gov

***Abstract*—Hardware Accelerators are playing increasingly important roles in achieving desired performance from desktop to cluster computing. While General Purpose computing on Graphics Processing Units (GPGPU) technologies have been widely applied to computing intensive applications, there is relatively little work on using GPUs and GPU-accelerated clusters for data intensive computing that typically involves significant irregular data accesses. In this study, we report our designs and implementations of a popular geospatial operation called Zonal Histogramming on Nvidia GPUs. Given a zonal dataset in the form of a collection of polygons and a geospatial raster that can be considered as a 2D grid, for each polygon, Zonal Histogramming computes a histogram of the values of raster cells that fall within the polygon. Our experiments on 3000+ US counties (polygons) over 20+ billion NASA Shuttle Radar Topography Mission (SRTM) 30 meter resolution Digital Elevation Model (DEM) raster cells have shown that, an impressive 46 seconds end-to-end runtime can be achieved using a single Nvidia GTX Titan GPU device. The runtime is further reduced to ~10 seconds using 8 nodes on ORNL's Titan GPU-accelerated cluster. The desired high performance opens many possibilities for large-scale geospatial computing that is important for environmental and climate research.**

Keywords - Zonal Histogramming, Geospatial Rasters, Point-in-Polygon Test, Parallel Computing, GPU

I. INTRODUCTION

Progresses in remote sensing technologies and increasing global and regional climate modeling have generated large amount of geo-referenced raster data with high spatial, temporal, spectral or thematic resolutions. Very often these large-scale raster data needs to be aligned to different types of zones, such as administrative and ecological regions, to derive per-zone histograms to understand the distributions of a specific environmental variable better in the zones. These histograms can further be used as feature vectors for more sophisticated analysis, such as computing various distance measurements which can be used for subsequent clustering. Such zone-based histogramming can be treated as an extension to traditional Zonal Statistics processing [1], a geospatial operation that has been extensively supported in various Geographical Information Systems (GIS) and remote sensing software packages where only major statistics, such as min, max, average, count and standard deviation, are reported as a table with each row corresponds to a zone. While the performance of the Zonal Statistics operations (including zonal histogramming) is typically acceptable for small datasets, significant performance issues may arise for zonal histogramming on large-scale rasters and complex polygons.

In this study, we aim at developing a new technique that can make full use of GPU accelerators to speed up Zonal Histogramming in both personal and cluster computing environments. Given a raster and a polygon input layers, our technique has four steps in computing the histograms of raster cells that fall within polygons. Each of the following four steps are mapped to GPU hardware by identifying its inherent data parallelisms (1) dividing an input raster into tiles and compute per-tile histograms, (2) pairing raster tiles with polygons and determining inside/intersect raster tiles for each polygon, (3) aggregating per-tile histograms to per-polygon histograms for inside raster blocks, and (4) updating polygon histograms for raster cells that are inside respective polygons through point-in-polygon test by treating raster cells in intersecting raster tiles as points.

Experiment results have shown that our GPU-based parallel Zonal Histogramming technique on 3000+ US counties (polygonal input) over 20+ billion NASA Shuttle Radar Topography Mission (SRTM) 30 meter resolution Digital Elevation (DEM) Model raster cells [2] has achieved impressive end-to-end runtimes: 46 seconds on a low-end workstation equipped with an Nvidia GTX Titan GPU. Furthermore, using the same test datasets, our approach has achieved 60-70 seconds using a single computing node and about 10 seconds using 8 nodes on the Titan supercomputer located at the Oak Ridge National Laboratory (ORNL) [3]. The results clearly demonstrate the potentials of using massively data parallel GPU accelerators for large-scale geospatial processing and can serve as a concrete example of designing and implementing popular geospatial data processing techniques on new parallel hardware to achieve desired performance.

The rest of the paper is arranged as the following. Section 2 introduces background, motivation and related work. Section 3 presents the set of parallel designs that are capable of utilizing GPU accelerator processing power for Zonal Histogramming and their realizations on GPUs and cluster computers. Section 4 provides experiment results on NASA SRTM 30 meter resolution DEM data on both a

single workstation and multiple computing nodes on Titan GPU-accelerated cluster. Finally Section 5 is the conclusion and future work.

II. Background, Motivation and Related Work

Geospatial processing has been undergoing a paradigm shift in the past few years. Advancements in remote sensing technology and instrumentation have generated huge amounts of remotely sensed imagery from air- and space-borne sensors. In recent years, numerous remote sensing platforms for Earth observation with increasing spatial, temporal and spectral resolutions have been deployed by NASA, NOAA and the private sector. The next generation geostationary weather satellite GOES-R (whose first lunch is scheduled in 2016) will improve the current generation weather satellite by 3, 4 and 5 times with respect to spectral, spatial and temporal resolutions [4]. With a temporal resolution of 5 minutes, GOES-R generates 288 global coverages everyday for each of its 16 bands. At a spatial resolution of 2km, 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. While 30-meter resolution Landsat Thematic Mapper (TM) data is already freely available over the Internet from USGS [5], sub-meter resolution satellite imagery is becoming increasing available, with a global coverage in a few days [6]. Numerous environmental models, such as Weather Research and Forecast (WRF), have generated even larger volumes of geo-referenced raster model output data with different combinations of parameters, in addition to increasing spatial and temporal resolutions. For example, the recent simulation of Superstorm Sandy on National Science Foundation (NSF) Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA) has a spatial resolution of 500 meters and a 2-second time step running at 9120*9216*48 three-dimensional grid (approximately 4 billion raster cells) with a single output file as large as 264 GB [7]. Geospatial data at such large scales is well beyond the capacity of most commercial and open source GIS systems which are primarily designed for traditional uniprocessors based on serial algorithms.

On the other hand, mainstream computer architectures have also gone through major changes. The past few years have witnessed fast growing numbers of processor cores (or multi-core CPUs) in personal computing systems. Another technical trend is the emerging General Purpose computing on Graphics Processing Units (GPGPU) techniques that first appeared in 2007 when Nvidia released its Compute Unified Device Architecture (CUDA) computing platform for its many-core GPUs for general purpose computing [8]. While ASCI Red, the world's first supercomputer with 1 teraflops double precision floating point computing power is made up of 104 cabinets and occupied 1600 sq feet in 1996 [9], the latest Nvidia Tesla K40 GPUs released in late 2013 can be plugged into a computer or a cluster computing node as PCI-E peripheral devices with even higher computing power [10]. Despite that a few studies have exploited the computing power of GPU accelerators for large-scale geospatial processing (see [11] for a brief review), there are still considerable gaps between the parallel computing power on modern commodity hardware and the achievable performance that mainstream geospatial processing software can offer. Previous investigations on parallelization of geospatial operations in 1990s are either based on shared-nothing or shared-memory parallel computing model [12] and most of them relied on coarse-grained task-level parallelisms. More recently, MapReduce/Hadoop based techniques have attracted significant research and application interests [13,14]. Although Hadoop-based systems can achieve good scalability, they typically have low efficiency with respect to system resource utilization [15] and may not be able to achieve the desired high-performance. In this study, we aim at developing data parallel techniques for Zonal Histogramming that can scale across multiple computing platforms, including GPUs and GPU-accelerated clusters that are made of identical computing nodes equipped with GPUs. By extensively exploiting data parallelisms in geospatial processing, raster and polygon data can be chunked in flexible ways and mapped to parallel hardware.

It is clear that Zonal Histogramming is closely related to point-in-polygon test which has been extensively studied in computational geometry [16]. If we treat all raster cells as points, the coordinates of the corners or centers of raster cells can be computed easily and it is straightforward to perform Zonal Histogramming on top of point-in-polygon test. However, point-in-polygon test is typically expensive as the complexity is generally proportional to the number of polygon vertices for a single test. When the number of raster cells and/or the number of polygon vertices are large, it would be inefficient to perform such test on all or even a subset of raster cells. In spatial databases, an operation similar to Zonal Histogramming is a Spatial Join [17] based on point-in-polygon test. To process such spatial joins efficiently, a common practice is to index both points and polygons so that only neighboring points and polygons are paired up in the Spatial Filtering phase before point-in-polygon tests are actually applied in the Spatial Refinement phase [17]. As detailed in Section III, our approach essentially indexes geospatial raster tiles implicitly so that only raster cells in tiles that intersect polygon boundaries require point-in-polygon tests which results in significant savings of computation.

III. GPU-based Parallel Designs and Implementations

Given a Raster R with cell c_{ij} at row i and column j having an integer value v_{ij}, where $0<i<M$, $0<j<N$ and $0<v_{ij}<B$, and a collection of polygons P, for each polygon P_k, we want to derive a histogram H_k with B bins where H_k^b is the number of cells that geometrically intersect with P_k and $v_{ij}=b$. Our

technique has four steps and each step can be realized on GPUs in parallel. The overall procedure is illustrated in Fig. 1. While the details of each step will be provided in Sections III.A through III.D, we would like to note that Step 1 (lower-left of Fig. 1) is used to derive per-tile histograms, which is independent of the polygon dataset. Step 2 is designed for spatial filtering to pair up polygons with nearby raster tiles. In Step 3, histograms of tiles that are completely within polygons are added to the respective polygon histograms directly. Finally, point-in-polygon test is performed for all the cells in raster tiles that intersect with polygon boundaries and polygon histograms are updated accordingly.

A. Per-Tile Histogram Generation

After a raster is loaded into GPU memory, a natural way to generate per-tile histograms is to assign each raster tile to a GPU thread block (left part of Fig. 2). We consider the following two factors that may potentially impact system performance, i.e., tile size and counting approaches.

There are tradeoffs in determining tile sizes. Using a large tile size will require less memory to store per-tile histograms but is likely to generate more tiles that intersect polygon boundaries which subsequently require more point-in-polygon tests for all cells in these tiles. For the NASA SRTM case study presented in Section IV, we empirically set the tile size to 0.1 by 0.1 degree. As such, given that SRTM DEM data has a spatial resolution of approximately 30 meters (1/3600 degree), each tile has 360 cells along both latitude and longitude directions. As the majority of raster cells have values less than 5000 (elevation in meters), we set the number of histogram bins to 5000. For a 5 by 5 degree raster, using an integer (4 bytes) for a bin count, the memory footprint for all the per-tile histograms would be 50*50*5000*4 bytes = 50MB. This is acceptable as all GPUs used in our experiments have at least 5GB memory.

As shown in the code segment in the right part of Fig. 2, all the threads in a thread block work in parallel to first zero-out histogram bins (line 3) before updating counts in respective bins. As the number of threads in a thread block (e.g., 256) is typically smaller than the number of bins and the number of cells, the threads need to loop through the histogram bins (line 2) and raster cells (line 6) with a stride of *blockDim.x*, which is the number of threads in the block. We note that accessing to global variables *raw_d* and *his_d_raster*, which store the input raster tile and the output histogram for the *idx*th tile, is largely coalesced as neighboring threads access nearby array elements (line 4 and line 10). We could have forced d_TILE_SIZE, which is the tile size (360 in SRTM data) to be multiples of *blockDim.x* for even better memory access; however, we have decided to allow users to set it arbitrarily for better programmability. We also leave exploring the possibility of pre-sorting tile cells using a better ordering (e.g., Morton Code [18]) to preserve spatial proximity and achieve better memory accesses (regardless whether the number of threads divides the number of columns in a raster tile) for future work.

For histograms with large numbers of bins (e.g., greater than 256), it is impractical any more to allocate a histogram to each thread for counting before the per-thread histograms are aggregated into a single per-block (i.e., per tile) histogram in a thread block. Given that the performance of atomic operations has been significantly improved in the latest Nvidia Kepler architecture, we have opted to use *atomicAdd* operator to simplify per-tile histogram generation (line 11), although more sophisticated techniques may potentially improve the performance.

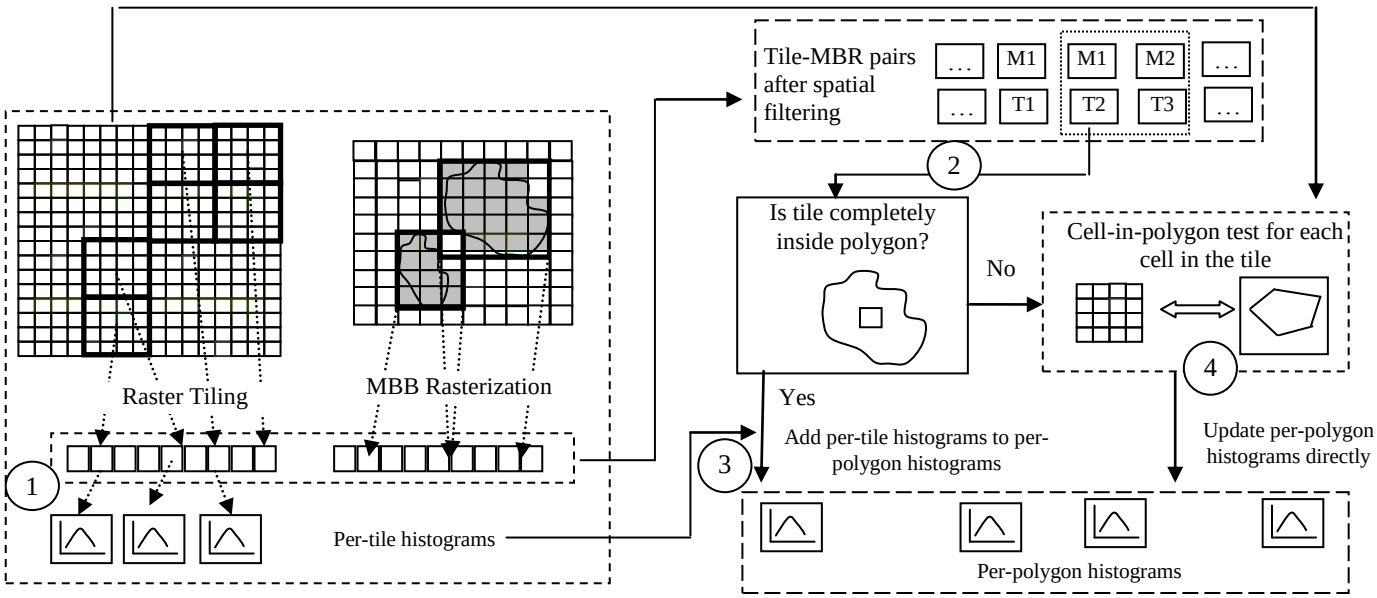
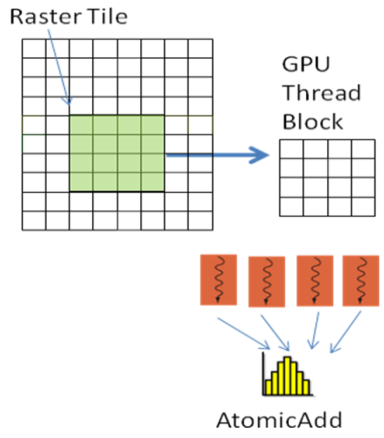


Fig. 1 Overall Design of Data Parallel Zonal Histogramming on GPUs




```
__global__ void CellAggrKernel(ushort* raw_d, uint* his_d_raster)
{
1       int idx=blockIdx.y*gridDim.x+blockIdx.x;
2       for(int k=0;k<HIST_SIZE;k+=blockDim.x)
        {
3               if(k+threadIdx.x>=HIST_SIZE) continue;
4               his_d_raster[idx*HIST_SIZE+k+threadIdx.x]=0;
        }
 5      __syncthreads();

6       for(int k=0;k<d_TILE_SIZE*d_TILE_SIZE;k+=blockDim.x)
        {
7               if(k+threadIdx.x>=d_TILE_SIZE*d_TILE_SIZE) continue;
8               int p=k+threadIdx.x;
9               //compute cell index based on p and store it in s
10              ushort v=raw_d[s];
11              atomicAdd(&(his_d_raster[idx*HIST_SIZE+v]),1);
12               __syncthreads();
         }
}
```

Fig. 2 Illustration and Code Segmentation for Step 1: Deriving Per-Tile Histogram

B. Pairing Raster Tiles with Polygons

The role of pairing raster tiles with polygons is similar to spatial filtering in spatial databases [17]. By observing that tiles in a raster can naturally serve as a grid-file for spatial indexing, we propose to reuse the GPU-based simple grid-file indexing technique that we have developed for point data. While we refer to [19] for design and implementation details, for the sake of completeness, we would like to reiterate the key points of the GPU-based data parallel pairing technique. The idea is to rasterize the Minimum Bounding Box (MBB) of all polygons according to the spatial tessellation of raster tiles. After each of the polygon MBBs are decomposed into a set of raster tiles, a polygon is paired with one or more raster tiles. Since MBBs are simple approximations of polygons, the relationship between a polygon and a raster tile can be one of the three cases: outside (0), inside (1) and intersect (2).

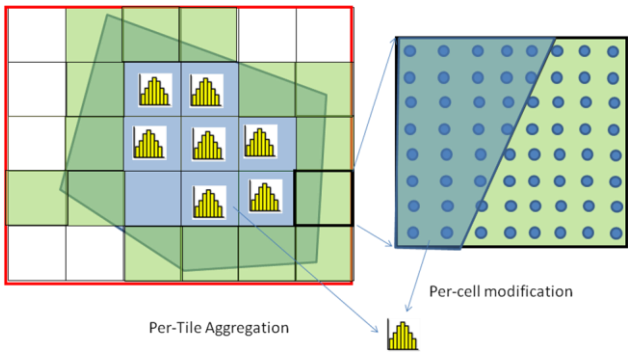


Fig. 3 Illustration of Integrating of Per-Tile Aggregation and Per-Cell Modification to Derive Per-Polygon Histogram

As shown in Fig. 3, we do not need to anything for tiles that are outside of the polygon being tested, as cells in these tiles should not be counted. For raster tiles that are completely within a polygon, we can simply combine the per-tile histograms into the per-polygon histograms. However, for raster tiles that intersect with the polygon, we will have to test whether a cell is inside a polygon and decide whether to count the cell in the per-polygon histogram, which is detailed in Section III.D. Although parallelizing tile-in-polygon test on GPUs is quite complicated as reported in our previous studies [20], this step (Step 2) typically incurs only a small fraction of overall runtimes. As such, practically, we can realize this step on CPUs using well-established computational geometry libraries and transfer the results back to GPUs for subsequent processing.

C. Aggregating Completely-Inside Per-tile histograms

Conceptually, Step 3 is the most embarrassingly parallelizable step among the four steps. However, given a array of tile-in-polygon test results with each element has a value of either 0 (outside), 1 (inside) and 2 (intersect), we need the following post-processing before the aggregation. First, the arrays of polygon MBB identifiers and raster tile identifies need to be sorted based on tile-in-polygon test results and polygon identifies so that all the tiles that are completely within a polygon become adjacent in the sorted arrays for better GPU memory accesses. Second, the numbers of raster tiles that are completely within polygons need to be counted. This can be realized by combining parallel primitives, including *stable_sort_by_key*, *stable_partition* and *reduce_by_key* provided in the Thrust library that comes with CUDA SDK, which can be simpler than using native parallel programming language such as CUDA directly. While we refer to [11] for more details on parallel primitives and their applications in geospatial computing, the left side of Fig. 4 illustrates the four parallel primitives by using a simple example.

For the GPU kernel that aggregates per-tile histograms, we assign a thread block to process a polygon. The kernel code segment is shown in the right part of Fig. 4 with arrays residing in global memory highlighted. For each block, we can retrieve the polygon identifier (*pid*), the number of raster tiles that completely fall within the polygon (*num*) and the starting position of the array that stores the raster tile identifiers (*pos*) based on the block identifier of the thread block (lines 3-5). The main body of the kernel code has two loops (lines 6-13). An outer loop processes all histogram bins in chunks with a stride of *blockDim.x*, i.e., number of threads in a thread block, in a similar way as we have discussed in Section III.A (line 6). Note that all threads in a thread block execute in parallel line by line. Threads are synchronized whenever there is a branch (e.g., "if" statement). For each thread, the inner loop (lines 10-13) iterates over the number of raster tiles that are completely within the polygon and adds the per-tile count to the per-polygon. It is clear that, before the outer loop in lines 3-5, all threads in a thread block access the same elements in the *pid_v*, *num_v* and *pos_v* arrays. As such, the global memory accesses can be coalesced to a maximum degree allowed by GPU hardware. In line 11, all threads will have the same *pos* and *i* values and access the same element in the *tid_v* array in a similar way. In lines 12 and 13, *w* is fixed for a particular *i* and *pid* is fixed across the thread block. The only changing variable in accessing the raster cell array (*his_d_raster*) and the output histogram array (*his_d_polygon*) is *p* which is calculated by adding thread identifier (*threadIdx.x*) to the chunk offset of histogram bins (*k*). As such, neighboring threads access neighboring array elements in both *his_d_raster* and *his_d_polygon* arrays and ensure coalesced memory accesses.

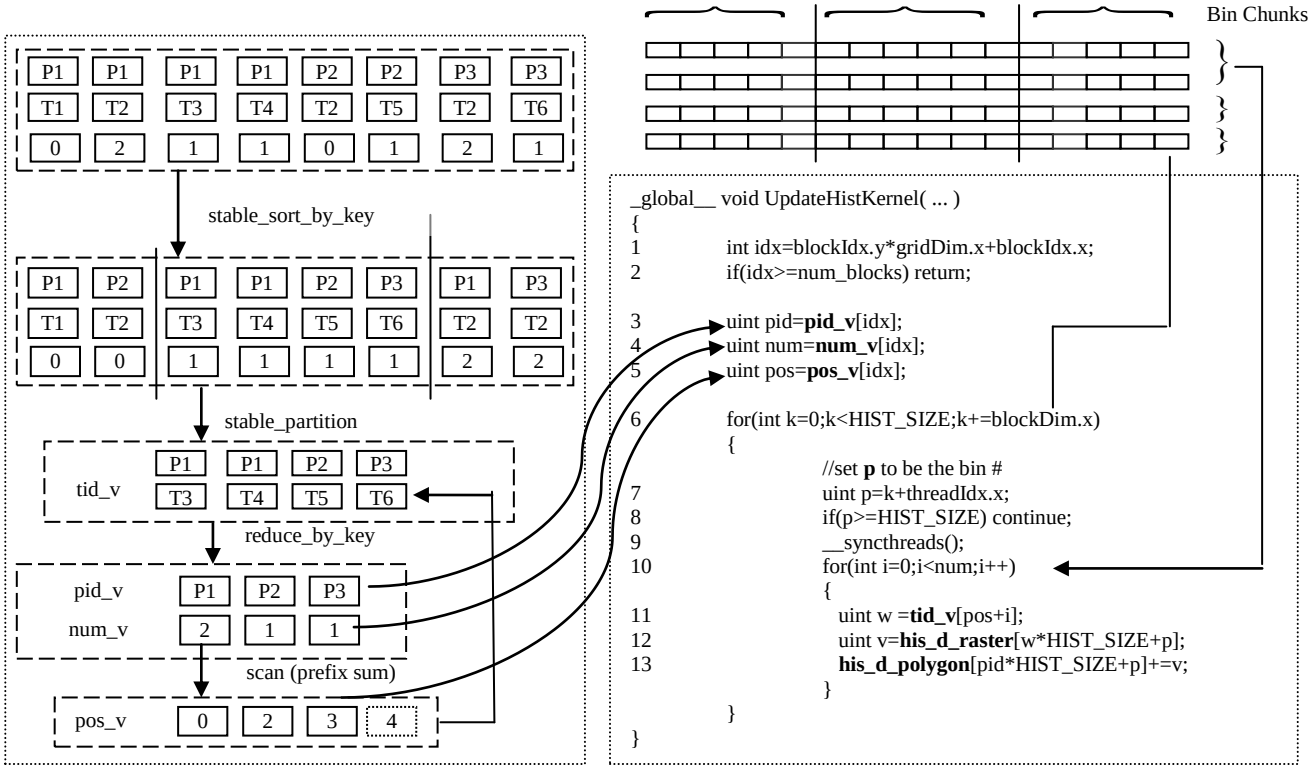


Fig. 4 Illustration of Deriving Polygon-Tile Pairs and Code Segment for Aggregating Per-Tile Histograms

D. Updating Per-Polygon Histogram

The last step might be the most computing intensive among the four steps. Here we reuse our GPU-based point-in-polygon test design [19] and adapt it for cell-in-polygon test. While we choose the center of a raster cell for point-in-polygon test for simplicity, it is possible to use some other points (e.g., corners or different types of weighted centers) either statically or dynamically that can represent the raster cell better, depending on applications. The code segment is shown in Fig. 5 by following a similar structure of Fig. 4. The middle-left part of Fig. 5 shows the relationship between the *ply_v* array and the *x_v* and *y_v* coordinate arrays. Note that the GPU-friendly array representation is significantly different from the popular object-based representation on CPUs. Basically the *ply_v* array indexes the *x_v* and *y_v* coordinate arrays. The beginning and the ending vertex index for polygon *k* are stored in *ply_v[k-1]* and *ply_v[k]-1*, respectively. The bottom-left part of Fig. 5 illustrates the basic idea of ray-crossing based point-in-polygon test [16]. If a line starting from the point being tested crosses the polygon boundary odd number times, the point will in the polygon; otherwise the point is outside of the polygon.

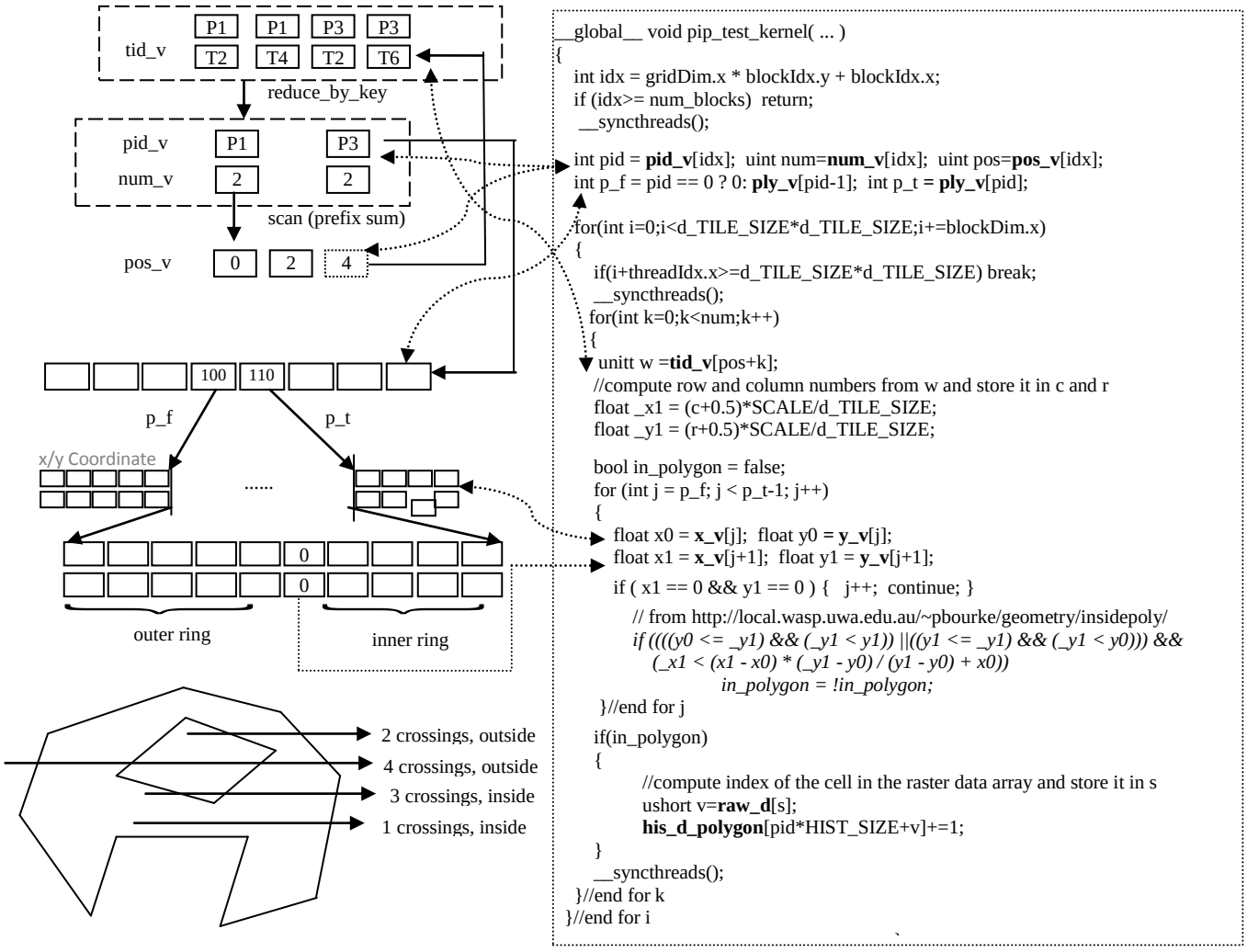


Fig. 5 Illustration of Identifying Intersecting Tiles and Ray-Cast Based Point-In-Polygon Test (Left) and Code Segment for Per-Cell Modification of Per-Polygon Histogram (Right)

While the CPU serial implementation of the ray crossing based point-in-polygon test by Randolph Franklin (italicized in the right part of Fig. 5) only handles single-ring polygons, motivated by the fact that adding the coordinate origin to the polygon vertex array will handle multi-ring polygons correctly, we have modified our GPU implementation [19] to support multi-ring polygons which are not rare in the US county dataset that we use for our NASA SRTM data case study (Section IV). Similar to the arguments made in Section III.C, while accesses to global memory array *ply_v* (polygon index) are inevitably non-coalesced due to non-continuous *pid* values, accesses to global arrays *pid_v* (polygon identifier array), *num_v* (number of raster tiles array), *pos_v* (first tile index array) and *tid_v* (tile index array) are coalesced. This is due to the reason that all threads access either the same array element or neighboring array elements. Furthermore, since each thread is assigned to process a single raster cell in the inner *j* loop, all threads access the same elements in the *x_v* and *y_v* arrays. As such, memory accesses can be combined to a maximum degree allowed by GPU hardware. We could have loaded polygon vertices to GPU shared memory before looping through them by all threads. While this may reduce global memory accesses to a certain degree, as GPU shared memory is still a limited resource, doing so may reduce the scalability of the implementation, in addition to being more complex.

IV. Experiments and Results

A. Data and Experiment Environment Setup

As a popular geospatial operation, Zonal Histogramming can be applied to a variety of environmental and climate applications. In this study, we will use the boundaries of 3000+ United States counties as the polygon dataset that has 87,097 vertices in total. For the raster dataset, we will use NASA SRTM data in the Continental

United States (CONUS) region with 20 billion raster cells. The NASA SRTM elevation data at 30 meter resolution was obtained on a near-global scale in February 2000 from a space-borne radar system and has been widely used in many applications since then. The CONUS raster dataset has a raw volume of 40 GB and about 15GB when compressed in TIFF format in 6 raster data files (we refer them collectively as the NASA SRTM raster hereafter). To use multiple Titan nodes to process the raster data in parallel, we have further decomposed the original 6 rasters into 36 smaller rasters. The original raster sizes and their partitions are listed in Table 1. Since it currently infeasible to decompress TIFF images efficiently on GPUs, we reuse our Bitplane Bitmap Quadtree (or BQ-Tree [21]) technique to compress the raw data. The data volume is reduced to 7.3GB, i.e. ~18% of original size. More importantly, the compressed data can be easily decoded into tiles on GPUs [21] for subsequent zonal histogramming. While disk I/O is still significant when compared with computing which deserves further research in a high-performance computing setting, we assume all BQ-Tree compressed data resides in GPU memory for all experiments.

Table 1 List of SRTM Rasters and Partition Schemas

Raster #	dimension	Partition Schema
1	54000*43200	2*2
2	50400*43200	2*2
3	50400*43200	2*2
4	82800*36000	2*2
5	61200*46800	2*2
6	68400*111600	4*4
Total	20,165,760,000	36

We have set up three experiment environments to test the efficiency and scalability of the proposed technique. The first two environments are single-node configurations that use a Fermi (Quadro 6000) and a Kepler (GTX Titan) GPU device, respectively. Note that both devices have 6 GB GPU device memory. The third experiment environment is the ORNL Titan GPU-accelerated cluster. At the time of writing, the ORNL Titan supercomputer is the largest GPU-accelerated cluster in the world and the K20 GPU devices equipped on Titan are also based on the Kepler architecture. We have varied the numbers of computing nodes to be used on Titan from 1 to 16. The end-to-end runtime is reduced to 7.6 seconds when using 16 nodes which is already good enough from an application perspective. We did not count disk I/O times on Titan as data was very often cached in its file system during our experiments. The performance of Titan's file system also varied significantly due to uneven workloads. As such, we did not include disk I/O times in the two desktop settings either. However, the runtimes for the cluster experiment setting to be reported did include MPI communication times since we measured the wall-clock time at each node and will report the longest runtime among all the nodes as the wall-clock end-to-end runtime. We next report and discuss the experiment results under the three experiment settings.

B. Results on Single Node GPU devices

The runtimes of the four steps in Zonal Histogramming for both the Quadro 6000 and GTX Titan devices are listed in Table 2. For the purpose of completeness and better understanding of end-to-end runtimes, we have also included raster decoding times as Step 0 runtimes. The end-to-end runtimes are larger than the total of the runtimes of the five steps (listed in the second-last row in Table 2) due to data transfer times between CPUs and GPUs as well as times to write output to disks. From Table 2 we can see that, as expected, Step 4 on cell-in-polygon test is the most expensive steps on both the Quadro 6000 and GTX Titan devices, followed by Step 1 in computing per-block histograms. Both Step 2 and Step 3 are insignificant when compared to Step 1 and Step 4. Step 0 on raster decomposition takes about 20% of the end-to-end runtimes, although not dominate, is significant. However, we argue that, given that the BQ-Tree compressed raster volume has been reduced from 40GB to 7.3 GB and assuming that the sustainable data transfer rate between CPU memory and GPU memory is 2.5GB/s, the data compression technique can reduce the CPU->GPU transfer time from 8 seconds to about 3 seconds, which can largely offset the incurred raster decompression times (especially on the GTX Titan device) although data compression is mostly designed for reducing disk I/O overheads.

Table 2 List of Individual Step and Accumulated Runtimes (in seconds) on Two Types of GPUs

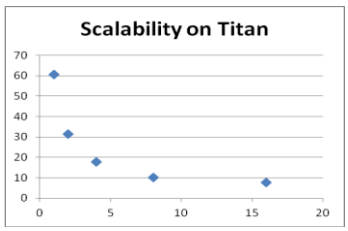
	Quadro 6000	GTX Titan
(Step 0): Raster decompression	16.2	8.30
Step 1: Per-block histogramming	21.5	13.4
Step 2: Block-in-polygon test	0.11	0.07
Step 3: "within-block" histogram aggregation	0.14	0.11
Step 4: cell-in-polygon test and histogram update	29.7	11.4
Runtimes of steps 0-4	67.7	33.3
Wall-clock end-to-end runtimes	85	46

It is also interesting to compare the runtimes on Quadro 6000 and GTX Titan where Step 4 is sped up 2.6X, Step 1 is sped up 1.6X and Step 0 is sped up nearly 2X. As a consequence, the end-to-end runtimes is nearly reduced to half on GTX Titan. This clearly indicates the advantage of the newer Kepler architecture on which the GTX Tian device is based. Compared with the previous generation Fermi architecture, on which the Quadro 6000 device is based, the Kepler-based GPU device not only has 6 times of processing cores (2,688 vs. 448, although Kepler cores have lower frequency) but also 2 times memory bandwidth (288.4 GB/s vs. 144 GB/s).

C. Results on Titan GPU-Accelerated Cluster

We used MPI for inter-node communications to make parallel designs scalable on GPU-accelerated clusters. The master node was used to combine per-polygon histograms (in case a polygon may intersect with multiple raster tiles) as this step only took a small fraction of a second. The end-to-end runtimes using 1-16 nodes on Titan are plotted and listed in Fig. 6. While we did not intend to compare our GPU-based implementation with existing GIS software as they are designed for different computing platforms and different scales of data, we have observed orders of magnitude better performance on a subset of the experiment data. While our tests stopped at using 16 Titan nodes as we had achieved the desired near interactive processing rate (in the order of seconds), we anticipate that our data parallel designs and implementations will scale with large datasets, as Fig. 6 suggests. We note that, when comparing the single node performance in the cluster computing setting (60.7 seconds) with that of GTX Titan (46 seconds), the 25% performance gap may potentially due to lower clock rate and bandwidth on K20 GPUs when compared with GTX Titan GPUs as well as MPI overheads.

It is also worthy of understanding that, as the number of nodes increases, each node processes a smaller number of tiles, which may bring inter-node load unbalance and reduce scalability. This is because, raster tiles that are at the edge of spatial coverage of polygon dataset, e.g., those in the southern part of Florida, are likely to have large portions of raster tiles that are completely outside of any polygon. As such, the work needs to be done in Step 4 for these tiles is much lighter than others. A potential improvement is to distribute the four steps in Zonal Histogramming to cluster nodes separately at the cost of more MPI communications. The tradeoffs between communication and load balancing need to be well studied to achieve high performance.



#of nodes	runtime (s)
1	60.7
2	31.2
4	17.9
8	10.2
16	7.6

Fig. 6 Plot of Runtimes (seconds) Against Number of Nodes on Titan GPU-Accelerated Cluster

V. Conclusions and Future Work

In this study, we report our parallel designs and implementations of several steps of the popular geospatial operation Zonal Histogramming on GPU accelerators. Experiments on both Fermi and Kepler GPUs have demonstrated impressive performance. Further experiments on ORNL Titan GPU-accelerated cluster have shown excellent scalability which makes the technique potentially useful to solve larger scale problems while achieving near real-time interactions on GPU accelerated clusters.

For future work, in addition to further improving single-node performance and achieving better load balancing on clusters as discussed in the relevant sections, we also would like to integrate the GPU-accelerated geospatial operation with visualization modules for interactive visual explorations. We also plan to design and implement more GPU-accelerated geospatial operations and help solve real world problems in efficient and scalable ways by using GPU-equipped workstations and GPU-accelerated cluster.

References

[1] D. M. Theobald (2005). GIS Concepts and ArcGIS Methods (2nd Ed.) Conservation Planning Technologies, Inc.
[2] http://www2.jpl.nasa.gov/srtm/
[3] http://www.olcf.ornl.gov/titan/
[4] http://www.goes-r.gov/
[5] http://landsat.usgs.gov/
[6] http://www.digitalglobe.com/
[7] Peter,J.,Straka, M., etc (2013).Petascale WRF Simulation of Hurricane Sandy Deployment of NCSA's Cray XE6 Blue Waters. Proceedings of ACM SC'13 Conference, #63.
[8] Kirk,B. and Hwu,W.-m. W. (2012). Programming Massively Parallel Processors: A Hands-on Approach, 2nd ed. Morgan Kaufmann.
[9] http://en.wikipedia.org/wiki/ASCI_Red
[10] http://www.nvidia.com/object/tesla-workstations.html
[11] Zhang, J. and You. S. (2013). High-Performance Quadtree Constructions on Large-Scale Geospatial Rasters Using GPGPU Parallel Primitives. International Journal of Geographical Information Sciences (IJGIS), 27(11), pp. 2207-2226
[12] Clematis,A., Mineter, M. and Marciano, R. (2003). High performance computing with geographical data. Parallel Computing, 29(10):1275–1279
[13] Aji, A., Wang,F. et al(2013). Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. Proc. VLDB Endow, 6(11), 1009--1020.
[14] Eldawy, A.,Li, Y., et al (2013). CG_Hadoop: Computational Geometry in MapReduce. Proc. ACM-GIS, 284-293.
[15] Lee, I.H., Lee, Y. J., et al. (2012). Parallel data processing with MapReduce: a survey. SIGMOD Record 40(4):11-20
[16] http://en.wikipedia.org/wiki/Point_in_polygon
[17] Jacox,E. H. and Samet,H. (2007). Spatial join techniques. ACM Transactions on Database Systems, TODS 32(1), #7.
[18] http://en.wikipedia.org/wiki/Z-order_curve
[19] Zhang, J. and You,S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Proc. ACM BigSpatial workshop.
[20] Zhang, J. and You, S. (2013). Parallel Zonal Summations of Large-Scale Species Occurrence Data on Hybrid CPU-GPU Systems. Technical report. Online at http://www-cs.ccny.cuny.edu/~jzhang/papers/zonalstat_tr.pdf.
[21] Zhang, J., You, S. and Gruenwald, L. 2011). Parallel quadtree coding of large-scale raster geospatial data on GPGPUs. Proc. of ACM-GIS, 457-460.

Acknowledgement: This work is supported in part by NSF Grant IIS-1302423. The work was initiated while Jianting Zhang was visiting ORNL through DOE Office of Science Visiting Faculty Program (VFP). We thank Dr. Yaxing Wei at ONRL for providing the NASA SRTM data. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. Oak Ridge National Laboratory is managed by UT-Battelle LLC for the Department of Energy under contract DE-AC05-00OR22725.