Speeding Up Geospatial Polygon Rasterization on GPGPUs

Jianting Zhang
Department of Computer Science
The City College of the City University of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

ABSTRACT

This study targets at speeding up polygon rasterization in large-scale geospatial datasets by utilizing massively parallel General Purpose Graphics Processing Units (GPGPU) computing for efficient spatial indexing and analysis based on a dynamically integrated vector-raster data model. As the first step, we have designed and implemented a parallelization schema for moderately large polygons using the Compute Unified Device Architecture (CUDA). Experiment results on 41,768 real world geospatial polygons with vertex numbers between 64 and 1024, which are selected among a total of 717,057 polygons with 1,199,799 rings in the experiment dataset, show that our implementation can speed up the computation of intersection points among polygon edges and scan lines by more than 20 times on a Nvidia C2050 GPU card. Extending the design and implementation to support polygons with arbitrarily large numbers of vertices by extensively using efficient sorting is discussed. The paper also reports the design and implementation of a profile quadtree to better understand the data and the distributions of its parallel computing tasks, in addition to help select polygon groups for experiments.

Keywords: Polygon Rasterization, Parallelization, GPGPU, Large-Scale

1. INTRODUCTION

Polygon rasterization is the process of converting vector polygonal data into gridded raster representation. While originating from computer graphics, rasterization has found its wide applications in geospatial data management and analysis. For example, rasterized polygons can be used for efficient spatial indexing in Spatial Databases [1] and fast geospatial analysis based on image/raster algebra [2]. It is a common practice to use hardware accelerations on GPUs to speed up rasterization and rendering in Computer Graphics. Despite the close relationships between Computer Graphics and GIS/Spatial Databases, the query driven geospatial applications are quite different from visualization driven computer graphics applications which makes it inconvenient, if not impossible, to use the fixed rasterization functionality of GPU hardware for geospatial data management and processing. Existing polygon rasterization modules implemented in open source GIS and spatial databases, e.g., GRASS [3] and GDAL [4], are based on the classic scan-line fill algorithms [5]. We have previously applied a modified GDAL rasterization implementation to build tree indices to facilitate managing a large collection of polygons representing the distributions of 4000+ bird species in the West Hemisphere [6]. Profiling the modules shows that calculating the intersection points between polygon edges and scan lines takes the majority of the processing time. It is thus desirable to speed up the polygon rasterization by efficiently utilizing parallel computing resources that are already available in commodity desktop computers.

The recently emerging General Purpose Graphics Processing Units (GPGPUs) computing technologies have provided a different set of programmable interfaces to support general purpose computing on GPUs. GPGPU technologies have been successfully applied in many areas [7] including our previous work on constructing quadtrees from large-scale raster geospatial data [8]. We aim at developing a parallelization schema and an efficient implementation for GPGPU-based software rasterization and quadtree construction for large-scale polygonal geospatial data. As the first step, we have parallelized the most costly step in the scan line algorithm on calculating the intersection points between polygon edges and scan lines. Besides introducing design and implementation details, we have performed extensive experiments on a large number of real world geospatial polygons in the Birds species distribution dataset.

The rest of this paper is arranged as follows. Section 2 introduces background and related works. Section 3 introduces the serial scan line fill algorithm and our research and development motivations. Section 4 presents the design and implementations of a preprocessing module to help understand both datasets and the spatial distributions of parallel computing tasks. Section 5 provides details on a preliminary design and implementation of GPGPU based parallel algorithm for computing intersection points along scan lines. Section 6 reports the experiment results and finally Section 7 is the conclusion and future works.

2. BACKGROUND, MOTIVATIONS AND RELATED WORKS

Rasterization is a vital step in computer graphics and numerous efforts have been put on developing efficient rasterization algorithms and implementations. While there are some previous works attempted to utilize graphics hardware to speed up spatial queries (e.g. [9]), unfortunately, a couple of issues prevent from using traditional fixed function graphics hardware interfaces for efficient and convenient geospatial data processing. First, rasterization algorithms on GPUs are usually proprietary and their hardware implementations largely remain black boxes. Some graphics hardware may be highly optimized for small triangles and do not support real world complex polygons, such as concave polygons or polygons with holes which are typical in real world geographical data. In fact, the GL_POLYGON primitive defined by OpenGL do not support concave polygons and is much slower than GL_TRIANGLES. While some tessellation and triangulation algorithms and packages are available to decompose complex polygons to simple polygons or triangles, they may not be supported by hardware and are left for serial software implementations. Second, to utilize GPU hardware acceleration before GPGPU technologies appear, researchers and developers in geospatial data processing are forced to transform geographical data, which are better presented as real numeric data types, into graphics-specific data types, such

ACM HPDGIS'11, November 1, 2011, Chicago, IL, USA.

as texture and colors, in order to get performance gains. Converting back and forth between geographical/projected coordinates and screen/view coordinates is not only cumbersome but also may lead to low accuracies. Third, many GPU APIs require a hardware context in runtimes. This makes the client/server architecture that is typically used in spatial databases impossible. While there are increasing graphics hardware supports for remote rendering, it is still not as convenient as in general purpose client/server computing. The GPGPU computing technologies have made massively parallel graphics hardware for general purpose computing possible which provides an exciting opportunity for software rasterization [10]. Compared with hardware rasterization, software rasterization is much more flexible and much easier to be tailored for geospatial applications.

Among the numerous advantages of using GPGPU computing technologies for processing polygonal geospatial data, the most attractive one to us might be the opportunity to develop a high-performance, dynamic and bidirectional conversion module between polygons and rasters through space-partitioning indexing trees (such as quadtrees) that support data compression and efficient query processing simultaneously. While conversion between vectors and rasters is a well-studied topic in GIS and Spatial Databases [11][12][2], to the best of our knowledge, most of the existing studies along the direction are based on serial computing framework. Massively parallel GPGPU computing technologies have made it possible to perform the conversion at the speed of rendering polygons to graphics displays and fast index polygonal data simultaneously. As the conversion and indexing, and, the subsequent raster-based analysis and visualization, can all be done in GPUs, expensive I/Os (including both data transfer between CPUs and GPUs and disk accesses) can be avoided. The combined parallel computing and I/O savings can potentially make quite some typical geospatial processing tasks orders faster. This in turn may enable turning traditionally offline geospatial modeling into interactive visual explorations. The uninterrupted visual explorations are likely to better facilitate scientific discoveries and decision making.

Efficiently utilizing GPGPUs for geospatial data processing in general and rasterization/indexing in particular have also imposed some signficant research and development challenges in realizing the potentials. First of all, GPGPUs have quite different hardware architectures and computing frameworks than traditional serial computing on uniprocessors. As the number of computing cores goes up, efficiently using on-chip shared memory and/or caches becomes crucial in achieving high performance. In addition, as the number of threads per core goes up, it is also crucial to coordinate the threads effectively to achieve high performance. Second, while GPGPUs work best for regular data structures such as vectors and matrices, it is necessary to make use of irregular data structures for large-scale geospatial data, such as tree indices, for efficient storage and query space pruning which may be better processed on CPUs in certain cases. Task/data partitioning among CPUs and GPUs and choosing between regular and irregular representations of geospatial data require considerable engineering efforts, including performance tuning and try-and-error.

While our ultimate goal is to develop an end-to-end high-performance system on modern commodity parallel hardware architectures for large-scale geospatial data processing, in this study, we will focus on speeding up computing intersection points between polygon edges and scan lines on GPGPUs using Nvidia's Compute Unified Device Architecture (CUDA) [13]. As detailed in Section 5, our current work is limited to rasterizing moderate sized polygons whose vertices can be fit into a GPGPU's computing block based shared memory. We are working toward rasterizing polygons with arbitrarily large numbers of vertices by dividing the vertices into groups and assembling the partial results. In this preliminary work, we also represent the rasterization results in the form of intersection point sequences with auxiliary data so that rasterization and quadtree construction can be efficiently performed subsequently as that has been done on CPUs in our previous work [14]. Despite that the aimed end-to-end system is still not fully developed, we hope this work can demonstrate the feasibility and potentials of software rasterization of large-scale real-world polygons in geospatial data processing. We next introduce some of the related works from both Computer Graphics and Spatial Databases communities.

Constructing spatial data structures and rasterizing triangles in computer graphics on GPGPUs have attracted considerable research interests in recent years. Zhou and his colleagues have implemented KD-Trees [15] [16] and octrees [17] on Nvidia GPUs using CUDA for large-scale triangles. The works are similar to indexing bounding boxes for primary filtering purposes in Spatial Databases [11] except these works are for 3D triangles in computer graphics applications instead of 2D polygons in geospatial applications. Our goal is more similar to the approach adopted by Microsoft SQL Server Spatial on rasterizing and indexing individual polygons [1] which provides finer level filtering capabilities. In addition, we also target at fast materializing polygon level indices to full rasters to speed up spatial analysis based on raster algebra [2]. There are also recent works on software-based rasterization of triangles on GPGPUs based on CUDA [10][18][19]. Considerable optimization techniques have been applied in these works, such as efficient utilization of fast shared memory, load balancing among computing blocks and reducing inter computing block communications, to improve the performance of software rasterization. According to the experiment results on a wide variety of test cases reported in [10], performance of software rasterization of triangles is within a factor of 2-8X compared to the hardware-based graphics pipeline on Nvidia GTX 480 GPUs. While the results are encouraging, it is unclear how to efficiently rasterize large-scale complex polygons on GPGPUs. Although authors in [19] attempted to use octrees to efficiently represent sparse rasters after rasterization, they all target at 3D graphics applications such as view dependent ray casting and shading which can not be applied to 2D geospatial analysis. Furthermore, we note that some of the optimization techniques in graphics rendering, such as early pruning based on visibility and 3D depth tests, generally can not be applied to overlapped geospatial polygons (if we consider different polygon datasets as the third dimension) as information on all polygons are needed to provide accurate query results in geospatial data processing. We also note that these works do not handle conversions between complex polygons and triangles and thus it is impossible to apply the works in our application directly.

Another category of closely related works is utilizing graphics hardware for data management in general and spatial databases and spatial analysis in particular. Since both vector and raster geospatial data has a spatial component, it is natural to perform certain spatial operations such as overlay and image convolution on GPUs. We refer to [9] for examples on spatial selections and joins using GPU hardware prior to GPGPUs. The GPGPU computing technologies have made geospatial data

processing on GPUs much easier and a detailed survey is beyond the scope of this paper. However, we refer to the works on multidimensional similarity joins [20] and density based spatial clustering [21] for examples that are relevant to spatial data processing. Our previous work on constructing min-max quadtrees from large-scale geospatial rasters has achieved a 23X speedup compared with serial CPU implementations [8]. Our recent work on decoding quadtree encoded bitplane bitmaps of large-scale geospatial rasters has achieved nearly 6X speedup when compared with a dual quadcore machine and 37X speedup compared with a single core [22]. This study is an expansion towards processing large-scale polygonal data on GPGPUs by incorporating our previous work on tree-based indexing of large-scale overlapped polygons on CPUs [14].

3 THE SERIAL SCAN-LINE FILL ALGORITHM AND MOTIVATIONS

Our GPGPU implementation of the classic scan-line fill algorithm [5] is based on the open source GDAL codebase [4]. Before presenting the details of the design and implementation in Section 4 (preprocessing a collection of polygons) and Section 5 (rasterizing a single polygon), we would like to introduce the scan line fill algorithm briefly. Given an enclosed polygon with n vertices where the 0^{th} vertex is the same as the $(n-1)^{th}$ vertex, we can construct n-1 edges from the vertices. We use the minimum and maximum values of the y coordinates of the polygon in the targeted raster tessellation, i.e., ymin and ymax, respectively, as the starting and the ending scan lines to compute the intersection points between all edges and all scan lines. For an edge (x1,y1,x2,y2) and a scan line y between y1 and y2, the intersection point can be easily calculated as (x1+(y-y1)/(y2-y1)*(x2-x1)).

According to the scan line fill algorithm, the intersection points of each scan line are sorted to generate interval pairs for subsequent polygon filling along the scan lines. While special cases, such as extreme vertices along y axis and horizontal edges, need to be handled separately, a simple polygon along a scan line usually have a pair of intersection points that define an interval of raster cells that are inside the polygon along the scan line. Complex polygons, such as polygon with holes, may generate multiple intervals along a scan line that need to be filled separately for rasterization. In addition to actually perform rasterization by filling the polygon cells, the y coordinate of the scan line and the x coordinates of the intervals are needed to construct a quadtree for the polygon which has been left for future work. For the example shown in Fig.1, there are 7 vertices and 6 edges in the polygon. Given a raster tessellation of 8 by 8, there are 7 out of 8 scan lines that intersect with at least one of the edges. For scan line y=2, two intersection intervals along the scan line, i.e., cells marked by circles, can be derived.

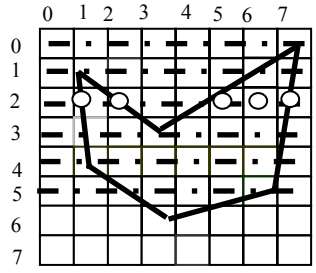

Fig. 1 An Example of the Scan Line Fill Algorithm

Experiments on the serial scan line fill and quadtree construction algorithms in our previous implementation [14] have shown that computing the intersection points dominates the whole serial process. A further analysis shows that the complexity of calculating intersections points is in the order of (n-1)*(ymax-ymin) where n is the number of polygon vertices. The computation is costly for large polygons with a large number of vertices and scan lines. It is thus desirable to parallelize the intersection computation process. While also refer to our previous work [14] on constructing quadtrees from pairs of intersection points derived from the scan-line algorithm. Parallelizing quadtree construction from intersection point pairs on GPGPUs are left for future work.

Although it is generally easier to parallelize the intersection point computation process on CPUs based on a few mature parallel abstractions, such as OpenMP, Intel Thread Building Blocks (TBB) and Microsoft Parallel Pattern Library (PPL), we have chosen GPGPUs for two reasons. First, GPGPU represent a new parallel computing paradigm that is drastically different from existing multicore CPUs although more recent multicore architectures such as Intel Many Integrated Core (MIC) fill the gap in between to a certain extent. Exploring the parallel computing power of GPGPUs with hundreds of processing cores and tens of thousands of threads is both challenging and rewarding. Second, GPGPUs usually work as accelerators for CPUs and can work with multicore CPUs to provide higher parallelization levels on commodity desktop computers.

To support load balancing and efficiently parallelize computations of intersection points, we have developed a preprocessing module that serves the following three purposes: (1) Assemble polygon vertices from disks and reside them in main-memory in a cache-friendly manner. (2) Derive metadata from the polygons and spatially profile the polygons. (3) Group the polygons into subsets so that they can be processed in parallel with balanced workloads. We next introduce the preprocessing module before introducing the GPGPU design and implementation in Section 5.

4 PREPROCESSING POLYGON COLLECTIONS

We assume a polygon collection consists of N datasets and all datasets follow Open Geospatial Consortium (OGC) Simple Feature Specification [23]. According to the specification, a polygonal feature may have multiple rings and each ring consists of multiple vertices. As such, we can form a four level hierarchy from a data collection to vertices, i.e., dataset → feature → ring→ vertex. Our first step in the preprocessing is to assemble polygon vertices residing on disks into continuous array data structures in main memory. Compared with using dynamic data structures that depends on pointer chasing, the array data structures are more cache friendly on CPUs. More importantly, as currently CPUs and GPUs use disparate memory spaces, array copying is the primary approach to efficiently transfer data in CPU memories to GPU memories.

Five arrays are used for a large polygon collection. Besides the x and y coordinate arrays, three auxiliary arrays are used to maintain the position boundaries of the aforementioned hierarchy. As shown in Fig. 2, given a dataset ID (0..N-1), the starting position and the ending position of features in the dataset can be looked up in the feature index array. For a feature within a dataset, the starting position and the ending position of rings in the feature can be looked up in the ring index array. Similarly, for a

ring within a feature, the starting position and the ending position of vertices belong to the ring can be looked up in the vertex index array. Finally, the coordinates of the ring can be retrieved by accessing the x and y coordinate arrays. It is easy to see that retrieving coordinates of single or a range of datasets, features and rings can all be done by sequentially scanning the five arrays in a cache friendly manner. It is also clear that the number of features in a dataset, the number of rings in a feature and the number of vertices in a ring can be easily calculated by subtracting two neighboring positions in the respective index array. As such, the array representation is also space efficient. The implementation of polygon assembling is on top of the GDAL open source library that has provided convenient accesses to many types of vector data formats, such as ESRI Shapefile and PostgreSQL/PostGIS databases. The assembling step simply loops through all the datasets and collect features, rings and vertices. The position indices of features, rings and vertices are advanced while the vertices are copied onto the two coordinate arrays.

The second step in our preprocessing procedure is to build a quadtree to spatially profile all polygons in a collection. We thus term the resulting tree index data structure as profile quadtree. While polygons in a single layer (dataset) are usually spatially disjoint, polygons across layers (datasets) can be significantly overlapped. This has made classic spatial indexing approaches, such as R-Trees or quadtrees [11][12], inappropriate for profiling purpose even if the index data structures are created for individual datasets (see [14] for more discussions). We have developed an enhanced quadtree to index multiple overlapped datasets by recording both spatial and thematic information of polygons (or features). Each leaf node of the quadtree contains a set of polygons whose MBRs (in the format of quadruples of (iminx, iminy, imaxx, imaxx)) are covered by the spatial extent of the node but are not covered by the spatial extent of any of its child node. In addition to the MBRs, dataset identifiers and feature identifiers of the polygons and the numbers of rings and vertices are also associated with the quadtree nodes. The procedure of constructing the profiling quadtree in pseudo C code is listed in Fig. 3. Note that all coordinates are integer values after snapping polygon vertices to the nearest raster cells based on a raster tessellation.

Estimating the computing workload for rasterizing polygons under a quadtree node can be performed as follows by using the profile quadtree. First, the number of scan lines can be calculated as (imaxy-iminy+1). Second, computation cost to calculate the intersection points of polygon edges and scan lines can be calculated as npoints*(imaxy-iminy+1). The total computing workload for the node can be derived by summing npoints*(imaxy-iminy+1) over all polygons associated with the node. The upper bounds for the resulting intersection points for a polygon can be estimated as the number of rings multiplied by the number of scan lines, i.e., nrings*(imaxy-iminy+1)*2*K where K is a constant factor to prevent overflowing. While the bounds are not needed in serial implementation on CPUs as the intersection points can be sequentially added to the output storage location, it is crucial for each processing unit to know its position to output so that processing units can work in parallel. Using upper bounds of output sizes can significantly save required memory footprints in parallel processing as otherwise large memory chunks would have to be allocated to all processing units.

Using the profile quadtree also make it possible to select subsets of polygons for further processing based on spatial and non-spatial criteria, such as number of vertices, number of rings, number of scan lines or their combinations. As detailed in Section 5, when processing a polygon using a GPGPU's computing block in our current implementation, the polygon's vertices must be fit in the block's shared memory. To select suitable polygons for GPGPU processing, we can traverse the profile quadtree to select polygons that satisfy the criteria.

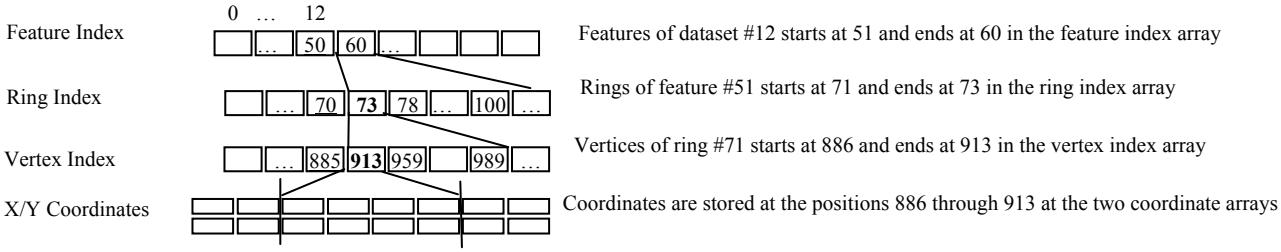

Fig. 2 Illustration of the Array Representation of Vertex Coordinates and Auxiliary Position Data

5 EFFICIENT POLYGON RASTERIZATION ON GPGPUS

We assume the general familiarity with the two-level (i.e., block and thread) parallel computing schema and the memory model (global memory, shared memory and local memory/registers) in CUDA. We also refer to the textbook [24] for introductions and the CUDA manuals available at [13] for more in-depth knowledge on CUDA and its programming.

We have chosen to assign polygons to computing blocks. In our design, the block level parallelization is enhanced with thread level parallelization by assigning edges of a polygon to threads in a computing block. While it is straightforward to assign polygons to threads and mimic the CPU serial implementation, we argue that the naïve parallelization schema design is inefficient for two reasons. First, launching a large number of threads on GPGPUs (e.g., 256 or more) within a computing block to process the same number of polygons requires a lot of registers and local variables for intermediate results. Unfortunately, even the latest CUDA Computing Capability 2.0 supports only a maximum of 32,768 registers per computing block. The number of registers is far fewer than what are required for complex computations like calculating intersection points in the scan-line fill algorithms. While it is possible to spill the register variables to global memory either programmatically or automatically by the complier, access to global memory is about two orders slower than to registers on Nvidia GPUs and thus the performance is likely to be unacceptable. Second, when computing the intersection points for all scan lines with all polygon edges, the vertices of the polygon in global memory would have to be accessed separately by all threads. This is likely to further worsen the performance.

In contrast, in our design, when a polygon is assigned to a computing block, the shared memory of the block can hold a polygon with reasonably sized vertices. For example, 8k shared memory can hold up to 1000 vertices whose x and y

coordinates are represented as integers or floats. By synchronized loading these coordinates to shared memory by all threads in a computing block, subsequent computing on intersection points does not need to access global memory any more which can significantly improve the overall performance. During the process of looping through all scan lines, we assign each polygon edge to a thread and compute intersection points independently. As only a few edges intersect with a scan line (c.f., the example in Fig. 1), the intersection result array is likely to be very sparse. It would take too much storage if the intersection result array is copied back to global memory directly for further processing. This could be very costly as well due to the slow accesses to global memory. Our solution is to perform a rank based compaction by moving the non-empty intersections to the front of the result array. Only the non-empty intersections are subsequently copied to the output array residing in global memory. The rank based compaction also utilizes shared memory and is very efficient. The pseudo CUDA kernel code based on the idea is illustrated in Fig. 4. We next further explain some technical details in the implementation.

```
typedef struct metatree
{
        unsigned char level;
        unsigned char xpos;
        unsigned char ypos;
        struct  metatree **children;
        struct  metatree *parent;
        std::vector<int> *rec;
} metatree;
```

```
void node2coord(const metatree * root, int& nx1,int& ny1,int& nx2,int& ny2) {
const metatree * temp=root;
nx1=ny1=nx2=ny2=0;
while(temp!=root) {
        nx1+=temp->xpos *xnum [temp->level];
        ny1+=temp->ypos]*ynum [temp->level];
        temp=temp->parent; }
nx2=nx1+xnum [croot->level];
ny2=ny1+ynum [croot->level];
```

```
void add_meta(metatree * root, int did, int fid, int iminx, int iminy, int imaxx, int imaxy, int nrings, int npoints){
        if (root->level==max_level)
                push back did, fid, iminx, iminy, imaxx, imaxxy, nrings, npoints to root->rec and return;
        call node2coord  to calculate the coordinates of the quadrant represented by root and store the values in nx1,ny1, nx2 and ny2
        if (iminx, iminy, imaxx, imaxxy) is completely within (nx1,ny1,nx2,ny2)){
                quad←-1;
                if(iminx,iminy,imaxx,imaxy) is completely within (nx1,ny1,(nx1+nx2)/2,(ny1+ny2)/2)) quad←0;
                if(iminx,iminy,imaxx,imaxy) is completely within (nx1,(ny1+ny2)/2,(nx1+nx2)/2,ny2)) quad←1;
                if(iminx,iminy,imaxx,imaxy) is completely within (nx1+nx2)/2,ny1,nx2,(ny1+ny2)/2)) quad←2;
                if(iminx,iminy,imaxx,imaxy) is completely within ((nx1+nx2)/2,(ny1+ny2)/2,nx2,ny2)) quad←3;
                if(quad>=0) {
                        if(root->children==NULL) allocate memory for root->children and initialize.
                        if(root->children[quad]==NULL) allocate memory for root->children[quad] and initialize
                        croot->children[quad]->xpos=quad/2;  croot->children[quad]->ypos=quad%2;
                        add_meta(croot->children[quad], did, fid, iminx, iminy, imaxx, imaxy, nrings, npints);
                }
        }
         else push back did, fid, iminx, iminy, imaxx, imaxxy, nrings, npoints to root->rec and return
}
```

Fig. 3 Data Structure and Procedures to Construct Profile Quadtree

```
//input: xx and yy are x and y coordinate array, respectively
//input: para is the parameter array with the following info for each polygon: MBR, nring, npoints, starting and ending positions of the polygon vertices on xx and yy arrays (ps and pe, respectively).
//output: triples is the array storing the resulting intersection points as the triples of (y coordinate of scan-line, number of intersection points, list of the x coordinate of intersection point)
//output: nums is the array storing the numbers of output triples for all polygons; if nums[i] is greater than the size of triples for polygon I then not all intersection points are output and recalculation is needed
__global__ void rasterize(double *xx, double *yy, int *para, int *triples, int *nums){
        __shared__  double padfX[MAX_PT]; //x coordinate array on shared memory
        __shared__  double padfY[MAX_PT]; // y coordinate array on shared memory
        __shared__  int polyInts[MAX_PT]; //temporary intersection result array  for one scan-line
        __shared__ int iminx, imaxx, iminy, imaxy, ps, pe, npoints; //metadata for the polygon being processed
        __shared__ int  ni, nv; // temporary variables used by the last thread to copy intersection result to global memory
        int tidx = threadIdx.x; // thread id
        int bidx = blockIdx.x; // block id
Step1:  copy values of iminx, imaxx, iminy, imaxy, ps, pe, npoints, from para[bidx] to the shared variables and set nv to 0, all by thread 0;
Step2:  copy elements from ps to pe in the xx and yy arrays to padfX and padfY, respectively, by all threads
Step 3: for (int y=iminy; y <=imaxy; y++) {
        Step 3.1: calculate intersection points of scan line y and store the results at polyInts, by all threads
        Step 3.2: compacting polyInts by moving non-empty values before the empty values (by calling scan4 in Fig. 5), by all threads
        Step 3.3 sequentially copy y, number of intersections (ni) and the first ni elements in polyInts to triples and advance nv by (ni+2) if ni>0, all by the last thread in the computing block
        }
Step 4: set nums[bidx] to nv by the last thread in the computing block.
}
```

Fig. 4 Pseudo CUDA Kernel Code for Parallel Computing of Intersection Points on GPGPUs

Among the steps shown in Fig. 4, Step 3.1 on computing all intersection points is the most computing intensive one which has motivated us to seek GPGPU accelerations. We note that while it is also possible to assign threads to scan lines and let each thread loops over polygon edges from a parallel computing perspective, we choose to assign threads to polygon edges and have threads loop over scan lines. The decision is mostly due to the fact that the latter choice is more convenient to output the intersection points to global memory for subsequent rasterization or quadtree constructions as discussed in Section 2 and 3. Among the steps, Step 3.2 is least straightforward which may require further illustration. Basically, for each scan line, the intermediate array to hold the x coordinates of intersection calculation, i.e., polyInts, is cleared up with -1 which indicates empty intersection. After executing step 3.1 in parallel by all threads, the x coordinates of the intersection points for the scan line will be stored in a per-thread local variable. The values of the variables of all threads are further used to calculate the positions that the values should be written to in the polyInts array. This can be done by setting 1s for non-empty intersections and 0s for empty intersections followed by an exclusive prefix sum (or scan) as shown in Fig. 5. The values of the prefix sum results are the positions that non-empty intersection values should be output in the array holding the compaction results (which is polyInts in our case).

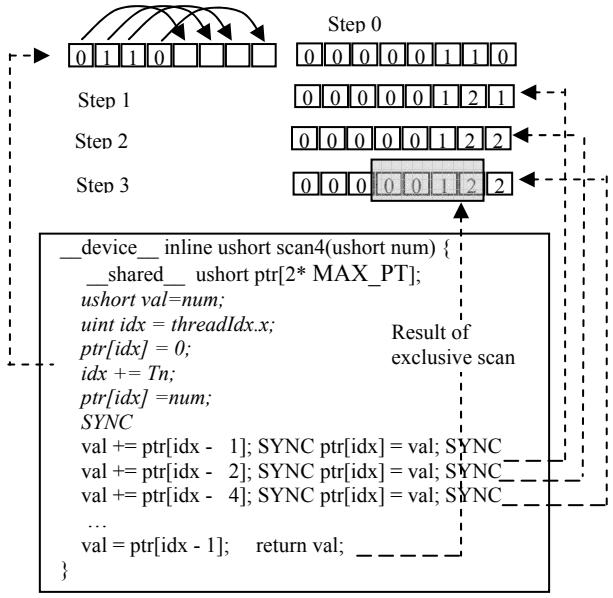

Fig. 5 An Example Illustrating Rank Based Compaction of Intersection Results on Shared Memory

Assuming the intersection result for the four threads in a computing block are (-1,4325, 4430,-1), as shown in Fig. 5, the initial values before the scan is thus (0,1,1,0) and the values after the scan would be (0,0,1,2). As such, 4325 and 4430 will be written to the index positions of 0 and 1 in polyInts by threads 1 and 2, respectively. The compaction process is essentially a type of sorting by assigning output positions to non-empty intersection points while skip empty intersection points. Prefix sum (or scan) can be efficiently performed on shared memory and the overhead is negligible. We note that, although the prefix sum implementation requires a temporary array of the size two times the size of maximum number of vertices (MAX_PT), it can be shared with the shared memory in the main kernel function (Fig. 4) which does not incur additional shared memory stress since scan4 is a device function.

6 EXPERIMENTS AND RESULTS

6.1 Data and Experiment Setup

We use the bird species distribution maps in the West Hemisphere from NatureServe [6] in our experiments. The dataset has also been used in our previous work on serial rasterization and quadtree construction on CPUs [14]. The dataset consists of ESRI Shapefiles for 4148 bird species with 717,057 polygons and 78,929,697 vertices. We do not exclude polygons with areas that are less than a cell as our previous works did. As a result, both the number of polygons and the number of vertices are slightly larger than we have reported previously. We have chosen five groups of polygons for experiments based on the number of vertices, i.e., (32, 64], (64,128], (128, 256], (256, 512] and (512, 1024], respectively. Although our implementation is capable of handling polygons with small numbers of vertices (e.g., below 32), we have found that it took only very little time to rasterize small polygons on CPUs and may not worth the overheads of GPU processing.

All the experiments are performed on a SGI Octane III machine. While the machine comes with two identical and independent nodes and four Nvidia Fermi C2050 GPU devices, only one node and a single GPU device on the node is used for the experiments. The computing node is equipped with dual Intel Xeon E5520 quadcore CPUs running at a 2.26 GHz clock rate, 48 GB 1333MHz DDR3 memory and 4 TB SATA 7200 RPM hard drives. The C2050 GPU card attached to the machine has 448 cores running at 1.15 GHz. The GPU device also has 3GB GDDR5 graphics memory running at 1.5 GHz clock rate. The GPU device is attached to the motherboard through a PCI-E x16 slot that can provide a theoretical unidirectional data transmit speed of 4GB/s.

Our primary measurement in this study is the wall-clock running times measured in milliseconds. We do not include data transfer times between CPUs and GPUs in the comparisons among CPU running times and GPGPU running times for two reasons. First, transferring polygon vertices and their auxiliary data from CPU to GPU is one time cost and is relatively insignificant compared to rasterization and subsequent spatial analysis. Second, we assume subsequent spatial analysis and visualization are all performed on GPUs which eliminates the need to transfer the processed data back to CPUs.

6.2 Results of Data Preprocessing

Assembling polygon vertices and deriving auxiliary position data took 447.3 seconds with cold cache while it only took 34.8 seconds with warm cache. Profiling results further show that the CPU time needed by the program (not including third party libraries such as GDAL) is only about 1.58 seconds. Given that the total volume of the geometry data (.shp files and .shx files) is about 1.3G, assuming an achievable 100 MB/s disk I/O rate, it requires only 13 seconds if the disk I/O bandwidth is fully utilized. As such, we suspect that the GDAL library that is used to access the shapefiles may be a major bottleneck. In contrast, the total data volume of the x and y coordinate arrays as well as the three auxiliary position arrays (whose sizes are listed in Table 1) is only about 609 MB and is less than half of the raw geometry data. It only took 30.9

seconds to load these five arrays from disks to main memory with cold cache (0.734 seconds with warm cache) and achieved a 14.4X speedup with cold cache. The signficant data loading speedup reflects a combined data volume reduction and using a simple linear data structure (array).

Table 1 Array Lengths of the Coordinate Arrays (1-2) and Auxiliary Position Arrays (3-5)

	Array name	Array Length
1	X coordinates	78,929,697
2	Y coordinates	78,929,697
3	Feature Index	4,148
4	Ring Index	717,057
5	Vertex Index	1,199,799

With respect to deriving the profile quadtree, we have used a 30 arc-seconds global raster tessellation which translates to a 21600*21600 grid for West Hemisphere. Our profile quadtree has 16 levels with a raster tessellation of 65536*65536 which is sufficient to cover the global extent. By traversing the profile quadtree, we can derive a few statistics to characterize non-empty quadrants such as number of polygons (ΣNP), number of vertices (ΣNV), number of scan lines (ΣNS) and number of intersections (ΣNI=Σ((NV-1)*NS). These statistics are then further aggregated based on profile quadtree levels. The results show that total number of edge-scan line intersection tests (ΣNI) is close to 200 billions which make it desirable to use GPU acceleration. The results also show that the majority of the tests are incurred by the large polygons associated with the top levels of the profile quadtree. Using the profile quadtree, by symbolizing the number of intersection tests in each quadrant, we can map the computing intensity spatially and understand both the dataset and its parallel computing tasks better. Fig. 6 shows the computing intensity map for level 6 with 99 quadrants using a rainbow coloring schema, i.e., red, blue and green indicate higher intensity while yellow and gray indicate lower intensity. The map is overlaid with the continental boundaries in the area for clarity.

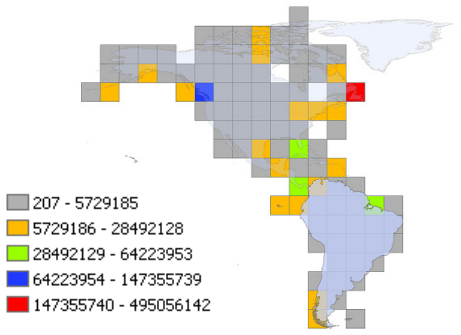

Fig. 6 Map of Spatial Distributions of Computing Intensities of the 99 Level 6 Quadrants in the Profile Quadtree

6.3 Comparison with Serial CPU Implementation

Table 2 lists the experimental results for the five polygon groups. From the results we can see that the GPGPU approach accelerates the processing times by about 20X for the four polygon groups with vertices between 64 and 1024 (41,768 polygons in total) when compared with the serial CPU implementation. The results are considerably signficant. On the other hand, the speedup for the polygon group with 32-64 vertices is only about 6X. Furthermore, it takes only half a second to compute the intersections of 46509 polygons in the group by the serial implementation. Considering the overhead of using GPGUP accelerations, the result may suggest that the advantages of GPGPU based rasterization may not be signficant for polygons with small numbers of vertices.

Table 2 Comparisons of Serial CPU and GPGPU Implements for Five Polygon Groups

Group #	1	2	3	4	5
Min # vertices	32	64	128	256	512
Max # vertices	64	128	256	512	1024
# Threads	64	128	256	512	1024
# Polygons	46509	23880	9666	5076	3146
CPU time (ms)	526	995	1803	4490	9387
GPU time (ms)	88	49	88	224	528
Speedup	6.0X	20.1X	20.5X	20.0X	17.8X

Based on our profile quadtree, there are 6960 polygons whose numbers of vertices are larger than 1024 and can not be processed by our current GPGPU implementation. While these polygons only count for less than 1% of the total number of polygons, our profile quadtree shows that they account for the majority of the intersection calculation workload where GPU acceleration could be most signficant. While our current GPGPU implementation does not have the capability of handling polygons whose numbers of vertices that are larger than the maximum number of GPU threads (1024), we next provide some discussions on how to extend the current implementation to handle polygons with arbitrary large numbers of vertices.

6.4 Discussions

The limiting factor of efficiently using GPGPUs to rasterize large polygons is the shortage of shared memory. The shared memory capacity is currently limited to 48K for the largest computing block with 1024 threads (i.e., a whole Stream Multiprocessor with 32 cores) on Fermi GPUs with Computing Compatibility 2.0. As the numbers of GPU cores and multiprocessors will grow significantly in the next few GPGPU hardware generations, instead of increasing, the shared memory capacity per-multiprocessor (and hence computing block) may actually decrease, in a way similar to multicore CPUs. As such, hardware advances will not solve the problem.

Our solution is to break the large numbers of vertices in large polygons into chunks and store the partial intersection results back to global memory so that the chunks of vertices can be processed independently. While the data decomposition strategy is common in parallel data processing for decades, the difficulty is how to combine the partial results into correct final results. Our idea is to expand the triple data structure in the form of (y coordinate of scan-line, number of intersection points, list of the x coordinate of intersection point) used in this study (see Section 5) to a set of triples in the form of (polygon identifier, y coordinate of scan-line, x coordinate of intersection point). These triples of computing block level results can be written to global memory in arbitrary order, including triples generated by different computing blocks for different vertices chunks. To assemble these triples and generate the interval pairs that are required for rasterization and quadtree constructions, a sort can be applied by using the combination of polygon identifier and y

coordinate of scan-line as the key and x coordinate of intersection point as the value. While any sorting algorithms that are available to GPGPUs can be used, a combined radix and merge sort is suggested as the triples are already partially sorted within blocks due to the vertex sequences. After sorting, all x coordinates of intersection points are contiguous for a particular scan line within a particular polygon which is exactly what we want as the starting point for subsequent rasterization and quadtree construction. We note that a few efficient sorting implementations are already available in CUDA [25][26].

7 CONCLUSION AND FUTURE WORK

In this paper, we have discussed a GPGPU accelerated software rasterization framework to rasterize and index large scale polygons. We have provided a GPGPU based design and implementation of computing intersection points which is the most expensive part of the classic scan-line fill algorithms. Experiments show that our implementation can achieve about 20X speedup for groups of polygons with vertices between 64 and 1024 using the birds species distribution data in the West Hemisphere that has about 3/4 million of polygons and more than 78 millions of vertices. We also have provided some design discussions on extending the current implementation to support polygons with arbitrarily large numbers of vertices by extensively using efficient sorting under different scenarios. Besides the preliminary GPGPU implementation of the scan line fill algorithms for real world geospatial polygons, we have also developed a profile quadtree that can be used to analyze and visualize spatial distributions of computation intensities in the context of computing the intersections of polygon edges and scan lines. The profile quadtree has also been used to guide select different polygon groups for experimenting the GPGPU software rasterization implementation.

The work reported in this paper is preliminary in nature as several important components in realizing a dynamically integrated vector-raster data model for high-performance geospatial analysis on GPGPUs are still currently under development. They certainly are the priorities on our future work list. First, as discussed in Section 6.4, we would like to extend our current implementation to support large polygons with arbitrary numbers of vertices. Second, we plan to implement the rasterization and quadtree construction based on the GPGPU derived triples. We keep it open on whether the final rasterization and quadtree construction step should be done on CPUs or GPUs. Third, once the fist two steps are completed, we plan to perform a comprehensive performance comparison with that of commercial spatial database indexing, such as Microsoft SQL Server Spatial [1], to demonstrate the benefits of parallel computing of large-scale polygonal geospatial data. Finally, we plan to develop indexing, query processing and spatial analysis engines based on the GPGPU codebase and integrate these backend engines with existing front modules in spatial databases, such as SQL parser and query optimizer, to provide an end-to-end, GPU accelerated, high-performance spatial database system.

REFERENCES

1. Fang, Y., M. Friedman, et al. (2008). Spatial indexing in Microsoft SQL server 2008. ACM SIGMOD conference, 1207-1216.
2. Theobald, D. M. (2005). GIS Concepts and ArcGIS Methods (2ed.). Conservation Planning Technologies, Inc.
3. GRASS GIS. http://grass.fbk.eu
4. GDAL: Geospatial Data Abstraction Library. http://www.gdal.org/.
5. Hearn, D. and M. P. Baker (1996). Computer Graphics, C Version (2ed). Prentice Hall.
6. NatureServe Digital Distribution Maps of the Birds of the Western Hemisphere. http://www.natureserve.org/getData/birdMaps.jsp.
7. Hwu, W.-M. W (eds.) (2011). GPU Computing Gems: Emerald Edition. Morgan Kaufmann.
8. Zhang, J., S. You, et al. (2010). Indexing large-scale raster geospatial data using massively parallel GPGPU computing. ACM-GIS Conference.
9. Sun, C., D. Agrawal, et al. (2003). Hardware acceleration for spatial selections and joins. ACM SIGMOD Conference, 455-466.
10. Laine, S. and T. Karras (2011). High-performance software rasterization on GPUs. ACM SIGGRAPH Symposium on High Performance Graphics.
11. Shekhar, S. and S. Chawla (2003). Spatial Databases: A Tour, Prentice Hall.
12. Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann.
13. Nvidia Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_home_new.html.
14. Zhang, J. (2009). Efficiently managing large scale species range maps in a spatial database environment. 17th International Conference on Geoinformatics.
15. Zhou, K., Q. Hou, et al. (2008). Real-Time KD-Tree Construction on Graphics Hardware. ACM Transactions on Graphics 27(5).
16. Hou, Q., X. Sun, et al. (2011). Memory-Scalable GPU Spatial Hierarchy Construction. IEEE Transactions on Visualization and Computer Graphics 17(4): 466-474.
17. Zhou, K., M. Gong, et al. (2011). Data-Parallel Octrees for Surface Reconstruction. IEEE Transactions on Visualization and Computer Graphics 17(5): 669-681.
18. Pantaleoni, J. (2011). VoxelPipe: a programmable pipeline for 3D voxelization. ACM SIGGRAPH Symposium on High Performance Graphics.
19. Schwarz, M. and H.-P. Seidel (2010). Fast parallel surface and solid voxelization on GPUs. ACM Transactions on Graphics. 29(6).
20. Lieberman, M. D., J. Sankaranarayanan, et al. (2008). A Fast Similarity Join Algorithm Using Graphics Processing Units. IEEE ICDE Conference: 1111-1120.
21. Bohm, C., R. Noll, et al. (2009). Density-based clustering using graphics processors. ACM CIKM Conference 661-670.
22. Zhang, J., S. You, et al. Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on Multicore CPUs and GPGPUs. Proceedings of the ACM-GIS Conference 2011.
23. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture. http://www.opengeospatial.org/standards/sfa.
24. D. B. Kirk and W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
25. Satish, N., M. Harris, et al. (2009). Designing efficient sorting algorithms for manycore GPUs. IEEE Symposium on Parallel & Distributed Processing.
26. Merrill, D. G. and A. S. Grimshaw (2010). Revisiting sorting for GPGPU stream architectures. ACM Conference on Parallel Architectures and Compilation Techniques, 545-546.