Constructing Natural Neighbor Interpolation Based Grid DEM Using CUDA

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, 10016
syou@gc.cuny.edu

Jianting Zhang
Dept. of Computer Science
City College of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

Abstract

Constructing digitial elevation model(DEM) from dense LiDAR points becomes increasingly important. Natural Neighbor Interpolation (NNI) is a popular approach to DEM construction from point datasets but is computationally intensive. In this study, we present a set of General Purpose computing Graphics Processing Unit(GPGPU) based algorithms that can significant speed up the process. Evaluating three real world LiDAR datasets each contains 6~7 million points shows that our CUDA based implementation on a NVIDIA GTX 480 GPU card is 1-2 orders faster than the current state-of-the-art.

Categories and Subject Descriptors D.2 [*Software*]: Software Engineering; H.2.8 [*Database Management*]: Database Applications—Data mining, Image databases, Spatial databases and GIS; I.3.1 [*Computer Graphics*]: Hardware Architecture—Graphics processors

General Terms Algorithms, Performance, Design

Keywords GIS, LiDAR, DEM, Natural Neighbor Interpolation, Voronoi Diagram, GPU, CUDA

1. Introduction

Light Detection And Ranging (LiDAR) technologies are widely used in many applications recently, such as terrain reconstruction and 3D modeling. Generally a LiDAR device collects data by scanning target objects with ultra-wave, laser or other light sources. Each scan generates a set of points with coordinates and other related information. The points of scanning are usually very dense and they are referred as point clouds. However, unstructured point clouds are not suitable for visualization and analysis in geographical applications. As such, Digital Elevation Models (DEM) need to be generated based on the scanned points through interpolations. The resulting DEMs are usually represented as raster grids where a height value (elevation) is associated with each grid cell.

To construct a grid based DEM for LiDAR dataset, *Natural Neighbor Interpolation*, or *NNI,*[13] is widely used for achieving better visual quality. A NNI process usually havs two steps: 1) generating a Voronoi diagram from a LiDAR point cloud. 2) performing grid queries on the resulting Voronoi diagram to derive a DEM. These two steps, especially for Voronoi diagram generation, could be computational intensive. In order to speed up the process, GPU based hardware accelerations can be adopted to parallelize both Voronoi diagram generation and grid queries as reported in[1, 2]

While previous works on GPU-accelerated NNI process are based on the traditional graphics APIs, in this study, we report our work on accelerating NNI process using GPGPU technologies, i.e., using GPU for general purpose computing. More specifically, we use NVIDIA Compute Unified Device Architecture (CUDA)[1] to program GPUs. Our approach is motivated by an efficient Voronoi diagram generation algorithm called Parallel Banding Algorithm (PBA) [3] but adopts a new parallelization desgin where well-established parallel primitives can be applied. Compared to TerraNNI, our approach is nearly an order faster in generating Voronoi diagrams and nearly two orders better in constructing grid DEMs. Furthermore, our approach does not require GPU hardware context which is more suitable for execution in a client-server environment. Compared to the original PBA implementation, our implementation is simpler and more portable while has a comparable performance.

The rest of the paper is organized as follows. Section 2 introduces CUDA background and related work on Voronoi diagram and NNI based grid DEM construction. Section 3 describes our proposed approach to generating Voronoi diagram and performing grid query for DEM construction based on GPGPUs. Section 4 provides implementation and experiment results on three real world datasets. Finally, Section 5 concludes the paper and briefly discusses future work directions.

2. Background and Related Work

2.1 GPGPU and CUDA

Graphics processing units(GPUs) are primarily designed for graphics applications such 3D games, and usually consist of very large amount of processors(i.e., GeForce GTX 480 has 480 cores). Modern computers and large clusters nowadays are equipped with GPUs to achieve higher performance. For example, in the list of Top 500 supercomputers[2], more and more supercomputers used GPUs to increase the floating-point operations per second(FLOPS). GPUs also provide personal workstations great performance improvements in very low price. For instance, a single GeForce GTX 480 GPU has a peek performance of 1.35 TeraFLOPS at the price of a several hundreds of U.S dollars currently. Zhang et al [14] discussed the advantages of GPU technology and proposed a framework of using GPUs

[1] CUDA: http://www.nvidia.com/object/cuda_home_new.html

[2] List of top 500 supercomputers: http://top500.org/

in high performance personal workstation in the context of geospatial computing. Although GPUs were designed as co-processors for graphics applications where they could only be accessed though graphics APIs (such as OpenGL[3]), modern GPUs have increasing support for general computations and can be manipulated using general programming languages such as C/C++. General Purpose Graphics Processing Units(GPGPUs) refer to GPU devices with general computing capabilities.

The Compute Unified Device Architecture (CUDA) from NVIDIA is a SDK and API for GPGPU programming. In CUDA programming model, there are three levels of execution units, i.e.,*thread*, *block*, and *grid*. Multiple threads are grouped together into blocks, where the threads in the same block can execute in parallel on *Streaming Multiprocessors*(SMs) and can access fast and shared block memory called *shared memory*. The highest level execute unit is called grid which consists of multiple blocks. A grid is a conceptual mapping of a CUDA function called *kernel* which can be invoked by CPUs. GPUs and CPUs have quite different hardware architectures. For CUDA-enabled GPUs, while a CPU core can only launch a small number of threads (usually 1 or 2), a GPU SM can simultaneously launch tens of thousands of threads. Unlike CPUs which can have very large, universally addressable memories and memories are organized into multiple levels of hierarchies, CUDA-enabled GPUs usually have a relatively large *global memory* (in the order of a few gigabytes) that can be accessed by all threads but a very small block level *shared memory* (in the order of a few kilobytes) that can only be accessed by threads within a block. While multi-level caching in CPUs is transparent to programmers, due to the speed gap (in the order of 100X) between accessing global memory and accessing shared memory, it is very important for developers to make full use of shared memory to achieve desired performance. It is also very important to balance between different GPU hardware resources in blocks including number of threads,shared memory and registers, to achieve maximum degree of parallelization. Because of the differences, a naive translation of a CPU based serial implementation into a GPU implementation usually results in poor performance. Despite tools are emerging to make CUDA based parallel programming easier, it remains non-trivial to design and implement parallel algorithms and implement them in CUDA.

2.2 Voronoi Diagram

Voronoi diagram is spatial decomposition of a given space based on a given set of points using a distance metric. In this paper, we consider 2D Voronoi diagrams. Given a 2D space $\mathbb{R}^2$ and a set of points $S = \{p_1 ... p_n\}$ (also called Voronoi sites), the space can be divided into a set of subspaces $\mathrm{Vor}_S(p)$ for each $p \in S$. The subspace, denoted as *Voronoi cell*, is defined as

$$\mathrm{Vor}(p) \quad = \quad \{x \in \mathbb{R}^2 \mid dist(x,p) < dist(x,q),\ \forall q \in S\}$$

where *dist* is Euclidean distance metric. By using a Voronoi diagram, given any point in the 2D space, we can find its nearest site by simply look up which Voronoi cell it belongs to. In practice, the 2D space is discretized into pixel representation, i.e., a matrix. The Voronoi diagram for such a 2D plane is a matrix where each element(pixel) is assigned to its corresponding Voronoi site. In this paper, we generally refer 2D discrete Voronoi diagram as Voronoi diagram when the context is clear.

The naive approach to generating a Voronoi diagram can be sequentially scanning the whole space and comparing the distances in each pixel to all the sites. The naive approach will compute a plane in $O(S(N-S))$ where S is the size of input sites and N is number of pixels in the plane. Fabbri et al. [4] reviewed and compared many state-of-the-art algorithms for Voronoi diagram computation. In order to further accelerate the process, GPU techniques are adopted in many works[3, 7, 10, 11]. A very early attempt was done by Hoff et al.[7]. They proposed a GPU based algorithm to compute Voronoi diagram using graphics APIs. For each input site, they rendered a right-angle cone in the image at a higher dimension and then projected back the rendering results. The distance test for each pixel was then performed using depth-testing provided by graphics APIs. In this approach, a 2D Voronoi diagram can be generated by rendering a 3D scene. Recent works[3, 10, 11] took advantage of general purpose GPUs that do not require a graphics context and achieved higher performance. Among those solutions, PBA[3] is an efficient design and implementation on GPGPUs using CUDA. PBA achieved linear computation time by using a dimensionality reduction approach and is able to compute exact Voronoi diagrams instead of approximate ones without losing performance.

2.3 Natural Neighbor Interpolation

Natural neighbor interpolation is a widely used spatial interpolation developed by [13]. Works reported in [1, 2, 5] discussed discrete and pixel based NNI in a 2D plane. The calculation of natural neighbor interpolation is based on a set of input points(sites) and their corresponding Voronoi diagram. The formal definition of natural neighbor interpolation is as follows, given a query point q in space $\mathbb{R}^2$, and a finite set of points S in $\mathbb{R}^2$ which are natural neighbors of q. Then natural neighbor interpolation query on q is defined as

$$v(q) = \sum_{p \in S} w_p v(p)$$

where w_p is the weight of p, v is the value of corresponding point(in our case is elevation value). Considering 2D discrete space, the weight of p for a given query point q can be calculated from

$$w_p(q) = \frac{Area(\mathrm{Vor}_S(p) \cap \mathrm{Vor}_{S \cup \{q\}}(q))}{Area(\mathrm{Vor}_{S \cup \{q\}}(q))}$$

where Vor is the Voronoi diagram based on the input sites, and $Area$ is the area of Voronoi cell. So w_p actually represents the "steal" portion from p's Voronoi cell to form the query point q's Voronoi cell. In the 2D discrete case where the space are represented as pixels, the continous space $\mathbb{R}^2$ can be simplified into pixel space $\mathbb{Z}^2$. The interpolation value for a query point q can then be further simplified as

$$v(q) = \frac{\sum_{c \in \mathrm{Vor}_{S \cup \{q\}}(q)} v(Site(c))}{Count(\mathrm{Vor}_{S \cup \{q\}}(q))}$$

where c is a pixel in q's Voronoi cell in the new Voronoi diagram, $Site(c)$ is site of original Voronoi diagram at c, and $Count$ outputs number of pixels for the input Voronoi cell. The interpolation computation now could be simplified as summing up height values of pixels and counting number of pixels for the Voronoi cell of q.

2.4 Related Works on NNI-based DEM Generation

As we discussed in Section 1, most GIS applications do not use points directly generated from LiDAR scanners. Instead of visualizing dense point cloud for the terrain surface, most GIS applications tend to use a raster representation called *digital elevation model*(DEM). DEMs are better for analysis and visualization and many spatial interpolation techniques have been used for generating DEMs. Mitas et al.[8] reviewed different spatial interpolation techniques. The linear interpolation is efficient but the visual quality is not good. On the contrary, Regularized Splines with Tension(RST) is well known to provide good quality but requires extensive computation. In order to produce better visual quality without suffering from the heavy computation overheads, Fan et al.[5]

[3] http://www.opengl.org/

proposed GPU assisted DEM construction using natural neighbor interpolation. The process of natural neighbor interpolation is accelerated by adopting GPU based Voronoi diagram generation proposed by Hoff et al[7]. Later, Beutel et al. [1, 2] extended the work of Fan et al. Their approach, termed as TerraNNI, used similar idea of Voronoi diagram generation used in [7]. The approach introduced region of influence for input sites which can handle gaps without adding pre-processing or post-processing steps. In addition, their approach improved performance of grid point interpolation by advanced the idea of batch queries used in [5]. However, their implementation still requires a graphics hardware context for the computation and additional post-processing step which they called *BufferAnalysis* for generating the final grid DEM. Unlike TerraNNI, our NNI implementation use CUDA directly which do not require graphics hardware contexts. This not only makes it more suitable for client-server based computing, where graphics hardware contexts are usually directly accessible to clients,but also significantly reduces data transfer between CPUs and GPUs due to graphics API invocations. We next present our designs of the Voronoi diagram generation and grid query algorithms on CUDA-enabled GPUs in the following two sections. Implementation details and experimental results are provided in Section 5.

3. Proposed Approach

Our proposed approach to DEM construction based on NNI has two components. The first component is to generate discrete Voronoi diagrams from LiDAR points using a raster representation and the second component is to query the Voronoi diagrams for each grid pixel and generate a DEM. During the process of generating a Voronoi diagram, the original LiDAR points are first rasterized into grid pixels, i.e., site pixels will have values of their coordinate indices and non-site pixels will be assigned to a special predefined value. Motivated by the good performance of PBA, we also adopt dimensionality reduction approach[3, 4] to generate Voronoi diagrams. Our approach has two steps: 1) Computing 1D Voronoi diagrams for all rows in the rasterized grid. 2) Assigning values for each column in a 2D Voronoi diagram using 1D Voronoi diagrams obtained from step 1. We next introduce our designs for the two steps as well as the grid query algorithm.

3.1 1D Voronoi Diagram

The Voronoi diagram for all the site pixels in a row is considered as a 1D Voronoi diagram. 1D Voronoi diagrams for all rows are processed in parallel and independently. Our design uses parallel scan primitives to generate 1D Voronoi diagrams. We believe using primitives can significantly reduce the design complexities and make implementations more portable. Furthermore, optimizing primitives naturally improves the efficiency of the implementations based on the primitives. Similarly, the implementations can be easily ported to other computing platforms that support the primitives.

To generate a 1D Voronoi diagram for each row, each pixel in the row needs to be assigned by its nearest site in its corresponding row. It is easy to see that the nearest site for a pixel in a 1D Voronoi Diagram is either the right-most site to its left (or left nearest site) or the left-most site to its right (or right nearest site). This step can be divided into two parallel scans, termed as *left_scan* and *right_scan*. The *left_scan* finds the left nearest site for each pixel and *right_scan* searches for the right nearest site for each pixel. The Voronoi site for each pixel can be derived by comparing the distances from the pixel to its left and right nearest sites. Suppose a pixel is $p_{i,j}$ where (i, j) is the index of current pixel, then its left nearest site $\mathcal{S}_{i,left}[j] = left\_scan(p_{i,0}...p_{i,j})$ and right nearest site $\mathcal{S}_{i,right}[j] = right\_scan(p_{i,j}...p_{i,0})$. By obtaining $p_{i,j}$'s left and right nearest sites, the pixel can be assigned by $min\_by\_dist(p_{i,j}, \mathcal{S}_{i,left}[j], \mathcal{S}_{i,right}[j])$, where operator *min_by_dist* returns the site which is closer to current pixel $p_{i,j}$ using Euclidean distance metric. The implementation of *left_scan* and *right_scan* can be directly derived from GPU parallel scan with minor modifications. More specifically, in the row based 1D Voronoi diagram where indices for pixels at each row is increasing from left to right, the *left_scan* is actually the maximum index of all sites appears before pixel $p_{i,j}$ on row i. In such case, we can set non-site pixels' column indices to a special negative value(*MIN_MARKER*) and then perform a scan on these values using operator *MAX*. On the other hand, the *right_scan* can be thought as the reversed scan using operator *MIN* and non-site pixels' column indices can be set to a large positive value(*MAX_MARKER*). The reversed scan can be realized by reversing the order of data layout in a regular scan. The 1D Voronoi diagram generation algorithm is shown in Algorithm 1. The work efficient parallel prefix scan on the GPU[12] can be adapted for the two scans.

Algorithm 1 1D Voronoi diagram

```
foreach row i do
    S_{i,left} ← left_scan(p_{i,0}...p_{i,n})
    S_{i,right} ← right_scan(p_{i,n}...p_{i,0})
    foreach pixel j do
        output[i, j] ←(i, min_by_dist(p_{i,j}, S_{i,left}[j], S_{i,right}[j]))
    end
end
```

3.2 Column Assignment

The column assignment step is to compute the final 2D Voronoi diagram based on 1D Voronoi diagram. Different from the first step that works on rows, this step works on columns. As discussed in PBA[3], the results of each column in the 1D Voronoi diagram generated in previous step are the candidates for column assignment.In PBA, candidate sets are maintained and pruned in a stack using doubly linked list. In contrast, instead of using complex data structures, our design works on pixels using parallel scans directly. We further divide the column assignment into two phases. The first phase is to generate a set of candidate sites for each column. And the second phase is to assign each pixel with its corresponding 2D Voronoi site coordinate using candidate set of each column.

Since columns are independent to each other, without losing generality we only consider one-column case. Assuming we are working on column $\mathcal{C}_j$, and $\mathcal{V}_j$ is the candidate set of nearest Voronoi sites for the column $\mathcal{C}_j$. A set of nearest sites($\mathcal{V}_j$) is a candidate set of nearest Voronoi sites for the column $\mathcal{C}_j$ if $\forall c \in \mathcal{C}_j,\ Site(c) \in \mathcal{V}_j$, where $Site(c)$ is c's corresponding Voronoi site. Initially, $\mathcal{V}_j$ contains all the values in column j of 1D Voronoi diagram generated from the previous step. In this case, there might be some sites in $\mathcal{V}_j$ which are not Voronoi sites for any elements in the column. In other words, those sites which are not Voronoi sites for column $\mathcal{C}_j$ and can be removed from $\mathcal{V}_j$ without breaking the definition of $\mathcal{V}_j$. We call a candidate set as *final* when no more sites can be removed from the set. The final candidate set for column j will be $\mathcal{V}_j^{final}$, which holds the property that for any point $p_{i,j}$ in column j and row i, there exists a unique element $v \in \mathcal{V}_j^{final}$ such that $dist(v,\ p_{i,j}) < dist(s,\ p_{i,j})$ where s is any Voronoi site except v and *dist* is Euclidean distance in this application. We denote the process of generating $\mathcal{V}_j^{final}$ from $\mathcal{V}_j$ as *candidate set pruning*. Before introducing our design for this step, we first show two properties that are used in our design.

Property 1. Suppose three sites $s_i(x_i,\ y_i)$, $s_j(x_j,\ y_j)$ and $s_k(x_k,\ y_k)$ with $y_i < y_j < y_k$ hold the relationship that $bisector(s_i,\ s_j)_{y=c} > bisector(s_j,\ s_k)_{y=c}$, where $bisector(a,\ b)$

is the intersection point[4] for perpendicular bisector of point a and b at column c, then s_j can be pruned by s_i and s_k at column c. This property we used is from [3]. For each site in the set of nearest sites, a pair of intersection y coordinates could be calculated using its left and right neighbor sites. We define the pair of intersection points as *intersection interval* for the corresponding site. And the intersection point between two sites $s_i(x_i, y_i)$ and $s_j(x_j, y_j)$ can be calculated by $\frac{(y_i+y_j)}{2} - \frac{x_j-x_i}{y_j-y_i} \cdot (x_j - \frac{x_i+x_j}{2})$. By introducing the concept of intersection interval, a site can be determined to prunable by comparing values in its corresponding intersection interval. For instance, if the intersection interval for site s_j is (y_m, y_n) where $y_m < y_n$, then s_j can be pruned by s_i and s_k, otherwise s_j is not prunable. Figure 1 is an illustration of this property.

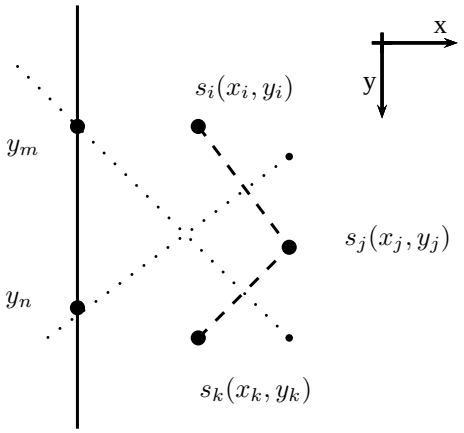

Figure 1. s_j is pruned by s_i and s_k. The intersection interval of s_j is (y_m, y_n).

Property 2. All the intersection points of the final candidate sites set $\mathcal{V}_j^{final}$ for column $\mathcal{C}_j$ are non-decreasing order with respect of increasing row indices. This property can be proved by contradiction. Assuming there exists an intersection point p for $\mathcal{V}_j^{final}$which breaks the property of non-decreasing order, and the corresponding site of p is $\mathcal{S}$. Then for $\mathcal{S}$, its intersection interval should be $(p,\ q)$ where $q < p$. Obviously, $\mathcal{S}$ can be pruned by its left and right neighbor using the Property 1 which leads to contradiction that $\mathcal{V}_j^{final}$ is final.

Inspired by Property 2, we have designed an iterative algorithm to prune the candidate set. In each iteration, we prune sites that do not hold Property 1 by calculating and checking the intersection interval. Property 1 specified in [3] provides an efficient approach only requires local neighbor information to prune a site from candidate site. Property 2 guarantees that the iteration is terminated in finite steps and at least one site can be pruned at one iteration. So for a candidate set with size N, the final candidate set can be obtained at most N iterations. Since each site's intersection interval is only relying on its left and right neighbor, the intersection intervals for all sites can be calculated in parallel. The calculation of intersection interval is introduced in Property 1 and denoted as *interval*. After checking the intersection intervals, we will know whether a site can be pruned or not. In order to keep track of this information, we use a flag array to represent the pruning status (0 for pruning and 1 for keeping) for the corresponding sites. With the help of flag array, the sites can be compacted by performing prefix sum on the flags. The whole algorithm is listed in Algorithm 2. *left* and *right*

[4] Since the whole column will have the same x coordinate, the intersect point will be represented by its y coordinate in our context.

Algorithm 2 Iterative Pruning

```
foreach column C_j do
    V_j^iter ← V_j  num_sites ← size(V_j^iter)
    repeat
        foreach site s(x_current, y_current) in V_j^iter do
            (p, q) ← interval(s_left, s_current, s_right)
            if p < q then
                flag[current] ← 1
            else
                flag[current] ← 0
            end
            index ← parallel-prefix-sum(flag)
            foreach site s(x_current, y_current) in V_j^iter do
                if flag[current] = 1 then
                    V_j^iter[index] ← V_j^iter[current]
                end
            end
            num_sites ← size(V_j^iter)
        end
    until num_sites does not change;
    V_j^final ← V_j^iter
end
```

Algorithm 3 Parallel Assignment

```
foreach column C_j do
    foreach site v in V_j^final do
        (start, end) ← interval(v)
        for index from start to end do
            output[index] ← v
        end
    end
end
```

functions are used for get left and right neighbors for the target site. The *interval* accepts three sites and produces the intersection interval for the center site. During the iteration, number of sites in $\mathcal{V}_j^{iter}$ is updated in order to terminate the loop.

The last phase is to assign sites for each pixel. We keep intersection intervals for last iteration in previous phase, so each site in $\mathcal{V}_j^{final}$ will be associated with an interval (y_m, y_n). With this information, we can located the corresponding pixels using the interval directly which leads to the parallel assignment algorithm in Algorithm 3.

3.3 Grid DEM Construction

Our grid DEM construction is based on natural neighbor interpolation for each grid point. We term the process of assigning value for each grid point as grid query. LiDAR instruments produce dense point cloud and elevation value for most locations. However, some objects such as lake can not be scanned by LiDAR devices(no elevation value will be returned). In order to take account of quality of the final DEM, we should also consider the gaps in the original data. As discussed in [1], these gaps are marked as "NO_DATA". Varies of methods can be used for processing these gaps, either in pre-processing, post-processing or during the interpolation. In our report, we use the same notation *region of influence* for input sites introduced by [1], which can handle gaps without additional preprocessing or post-processing. For a query point, the interpolation only uses the sites whose radius contain the query point. Similar to

Algorithm 4 Grid DEM construction based on NNI

```
foreach query point q do
    sum ←0 count ←0
    foreach pixel p in circle(q, r_query) do
        if dist(q, site(p)) < r_site and p ∈ Vor(q) then
            sum ←sum + v(p)
            count ←count + 1
        end
        if count > 0 then
            output[q]←sum/count
        else
            output[q]←NO_DATA
        end
    end
end
```

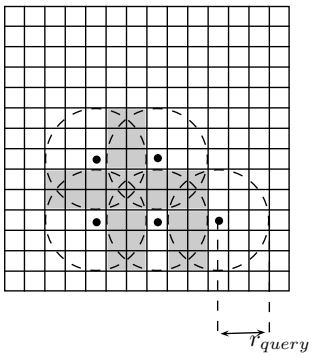

Figure 2. Grid Query

TerraNNI, we also restrict the influence size of query points using *query radius*.

Different from previous works that rely on rendering scenes on GPUs using graphics API to determine neighboring Voronoi sites for query points before the interpolation [1, 2], our design on grid query loops through the pixels within the query radius of the query points directly. As listed in Algorithm 4, our interpolation for the grid DEM construction uses only one single function which can be realized in one CUDA kernel. r_{query} and r_{site} are parameters for query radius and site influence radius in the algorithm. The condition $p \in \mathrm{Vor}(q)$ in the algorithm can be implemented by comparing distance of p and its Voronoi site and distance of p and q. Actually, the calculation of condition $p \in \mathrm{Vor}(q)$ is mathematically equivalent to draw the query cone in TerraNNI.

4. Implementation and Experiments

4.1 Implementation Details

Implementing the 1D Voronoi diagram algorithm on CUDA exploits two levels parallelism: rows are mapped to CUDA computing blocks and pixels within rows are mapped to CUDA threads. We have adapted a parallel scan implementation from Thrust[6] codebase for our implementation but other work efficient parallel scan implementations [12] can also be used. Directly derived from the Algorithm 1, our implementation contains two kernels each for a scan. For both kernels, we use number of rows as the number of blocks, and each block use 1024 threads. The parallel scan for a row is inside a block, so if the length of the row is larger than 1024, then the row will be grouped by size of 1024 and processed by 1024 threads serially. The comparison in the last step for the final output in Section 3.1 is combined with the second scan to reduce the overhead of lunching a new kernel.

Similar to 1D Voronoi diagram generation, we use one CUDA block for each column in the first part of column assignment, i.e., iterative pruning. If the size of the candidate set at a column is larger than the number of threads in the block (maximum 1024), a loop is then used. We have also tried to use shared memory to speed up accesses to GPU global memory in this phase but was not able to significantly improve the performance which lead us to suspect that the CUDA kernel for this step is computation bound rather than I/O bound. The second part on parallel assignment is currently implemented in a straightforward manner where each thread within a block is responsible for assigning pixels in an interval. A more load-balancing implementation (again using prefix sum for count-then-write) is being developed.

The implementation for grid query is straightforward by assigning each point query to a single thread. This step is obviously I/O bound and the classical stencil technique is used to reduce data accesses to GPU global memory by using GPU shared memory. As shows in Figure 2, five queries with query radius r_{query} are illustrated in the figure. The shadow pixels are those which will be used in multiple queries. Loading these pixels once in a memory-coalesced manner to serve for multiple queries can significant improve performance. However, the GPU shared memory is relatively small(48KB per SM in our GPU) and thus the number of threads in a block which can use the same shared memory relies on r_{query}. As r_{query} increases, number of threads per block will decrease. Not being able to fully utilize threads certainly will hurt the performance. Since the number of threads per block(η) can be determined by a configuration of r_{query} and size of shared memory, we propose using a threshold θ for number of threads per block to determine whether to use shared memory or not. If $\eta < \theta$, then using shared memory will achieve good performance.

4.2 Performance

We have evaluated our design and implementation using real world datasets and compared the results with those of TerraNNI. Our experiment environment is a workstation equipped with two Intel Xeon X5650 CPUs at 2.67GHz and 24 GB internal memory. The operating system for the machine is Windows 7 Professional 64-bit. We used NVIDIA GeForce GTX 480 graphics card running on CUDA 4.0 as our GPU platform. GTX 480 has 15 multiprocessors(480 CUDA cores in total), 1.4 GHz clock speed, 1.5 GB global memory and 48 KB shared memory per multiprocessor. According to the manual[9], GTX 480 can accommodate up to 8 active thread blocks per multiprocessor(or up to 1536 active threads / 48 active warps per multiprocessor), and each computing block can have up to 1024 threads. CUDA computing compatibility 2.0 with O2 optimization are used.

To evaluate the performance of our design and implementation, we have downloaded three LiDAR datasets collected by Florida Division of Emergency Management Coastal LIDAR Mapping Project[5]. We set the output grid size to 5000×5000, the influence radius of sites to 10, scale size to 1 and the query radius to 3 for all the experiments. As a comparison, we also tested the same datasets using the same parameters on TerraNNI[6]. For TerraNNI, both cone and plane settings are used and reported. Table 1 shows the results of experiments. One DEM result from our algorithm shows in Figure 3.

In Table 1, "Voronoi" refers to Voronoi diagram generation and "Query" refers to grid DEM query. We note that the running times

[5] http://mapping.ihrc.fiu.edu/fldemlidar20120119/Default.aspx.

[6] downloaded from https://github.com/thomasmoelhave/TerraNNI with minor changes to make it run on Windows.

		TerraNNI (Cone)		TerraNNI (Plane)		ScanIteration	
Datafile	# of points	Voronoi	Query	Voronoi	Query	Voronoi	Query
LID2007_075111_W	6982873	7.779	38.14	2.502	12.66	0.677	0.142
LID2007_075689_W	6066378	6.794	38.629	2.273	12.539	0.637	0.14
LID2007_073008_W	6211257	6.967	39.29	2.298	12.568	0.652	0.137

Table 1. Running time for the FL datasets (time in seconds)

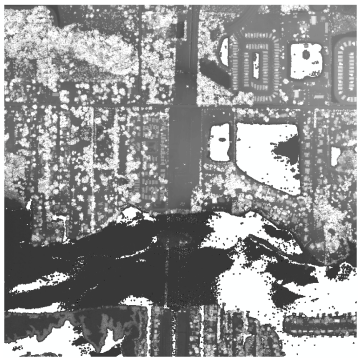

Figure 3. DEM generated by our algorithms

listed in Table 1 include data transfer times between CPU and GPU. As discussed in [1, 2], cones were approximated by k-gons for Voronoi diagrams generation and planes rendering were further used for achieving better performance. Our scan and iteration based implementation does not use approximation and the interpolation error will only be bounded by the grid discretization error. Compared with TerraNNI, even when plane approximation is used, our implementation outperforms TerraNNI on both Voronoi diagram generation(~3.5 times) and grid query(~88 times).

For Voronoi diagram, the performance differences between TerraNNI and our implementation might be due to the overheads of using GPU graphics APIs. Using CUDA to directly manipulate can better utilize GPU hardware resources, such as shared memory. For the grid query, the number of concurrent queries in TerraNNI is restricted by the graphics APIs. Batching queries in TerraNNI, while effective in improving parallelization, needs to save the results as intermediate representation which requires additional analysis to derive a final result. Furthermore, the number of queries at a batch in TerraNNI is determined by the size of query radius which limits its parallelism. In contrast, our implementation can handle arbitrary size of query radius and fully utilize GPU hardware resources.

5. Conclusion

This paper presents a set of GPU algorithms implemented using CUDA for constructing grid DEMs. The two parts of our proposed solution, including Voronoi diagram generation and grid queries, significantly outperforms TerraNNI which is a state-of-the-art software for NNI based grid DEM construction. We have also proposed a new approach to generating Voronoi diagram on GPGPUs. Our approach is simple, efficient and scalable by using parallel scan primitives. Meanwhile, our implementation is also scalable for large datasets.

For future work, we plan to further improve the efficiency of our design and implementation. For the iteration algorithm used for candidate sites pruning and the parallel assignment algorithm, we plan to investigate load balancing techniques and tailor them for GPGPUs. We also would like to evaluate our design and implementations on more and larger real world datasets to examine the scalability and efficiency of our design and implementations.

References

[1] BEUTEL, A., MØ LHAVE, T., AND AGARWAL, P. K. Natural neighbor interpolation based grid DEM construction using a GPU. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems* (New York, NY, USA, 2010), GIS '10, ACM, pp. 172–181.

[2] BEUTEL, A., MØ LHAVE, T., AGARWAL, P. K., BOEDIHARDJO, A. P., AND SHINE, J. A. TerraNNI: natural neighbor interpolation on a 3D grid using a GPU. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (New York, NY, USA, 2011), GIS '11, ACM, pp. 64–74.

[3] CAO, T.-T., TANG, K., MOHAMED, A., AND TAN, T.-S. Parallel Banding Algorithm to compute exact distance transform with the GPU. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 83–90.

[4] FABBRI, R., COSTA, L. D. F., TORELLI, J. C., AND BRUNO, O. M. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Comput. Surv. 40*, 1 (Feb. 2008), 2:1—-2:44.

[5] FAN, Q., EFRAT, A., KOLTUN, V., KRISHNAN, S., AND VENKATASUBRAMANIAN, S. S.: Hardwareassisted natural neighbor interpolation. In *In: Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX* (2005).

[6] HOBEROCK, J., AND BELL, N. Thrust: A Parallel Template Library, 2010.

[7] HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 277–286.

[8] MITAS, L., AND MITASOVA, H. Spatial interpolation. In *Geographical Information Systems: Principles, Techniques, Management and Applications*, P. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, Eds., vol. 1. Wiley, 1999, pp. pages 481—-492.

[9] NVIDIA. NVIDIA CUDA C Programming Guide 4.0, 2011.

[10] RONG, G., AND TAN, T.-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 109–116.

[11] RONG, G., AND TAN, T.-S. Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. In *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering* (Washington, DC, USA, 2007), ISVD '07, IEEE Computer Society, pp. 176–181.

[12] SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), GH '07, Eurographics Association, pp. 97–106.

[13] SIBSON, R. A brief description of natural neighbour interpolation. In *Interpreting multivariate data*, V. Barnett, Ed., vol. 21. John Wiley & Sons, 1981, ch. 2, pp. 21–36.

[14] ZHANG, J. Towards personal high-performance geospatial computing (HPC-G): perspectives and a case study. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems* (New York, NY, USA, 2010), HPDGIS '10, ACM, pp. 3–10.