Fast Filtering of LiDAR Point Cloud in Urban Areas Based on Scan Line Segmentation and GPU Acceleration

Xiangyun Hu, Xiaokai Li, and Yongjun Zhang

***Abstract*—The fast filtering of massive point cloud data from light detection and ranging (LiDAR) systems is important for many applications, such as the automatic extraction of digital elevation models in urban areas. We propose a simple scan-line-based algorithm that detects local lowest points first and treats them as the seeds to grow into ground segments by using slope and elevation. The scan line segmentation algorithm can be naturally accelerated by parallel computing due to the independent processing of each line. Furthermore, modern graphics processing units (GPUs) can be used to speed up the parallel process significantly. Using a strip of a LiDAR point cloud, with up to 48 million points, we test the algorithm in terms of both error rate and time performance. The tests show that the method can produce satisfactory results in less than 0.6 s of processing time using the GPU acceleration.**

***Index Terms*—Acceleration, fast filtering, graphics processing unit (GPU), light detection and ranging (LiDAR), scan line, segmentation.**

I. Introduction

LIGHT detection and ranging (LiDAR) integrates the Global Navigation Satellite System and Inertial Navigation System with laser scanning and ranging technologies. It enables direct measurement of the 3-D coordinates of points on ground objects for the efficient creation of digital surface models (DSMs). This massive set of points is called "point cloud." Modern airborne LiDAR technology can map the Earth's surface at a 15–20-cm horizontal resolution, and future generations of LiDAR scanners are expected to generate even higher resolution maps [1]. The large volume of scanned data that are manipulated when processing a LiDAR point cloud has been one of the major challenges in data processing. For example, one strip of a scanned area can easily reach tens of millions of points. Efficient algorithms are therefore important in practical applications. For some critical fields, such as emergency response, very short production time is required. For example, after an earthquake, terrain maps are required quickly for damage estimation and rescue plans.

Manuscript received February 29, 2012; revised May 5, 2012 and May 22, 2012; accepted May 25, 2012. Date of publication July 23, 2012; date of current version October 22, 2012. This work was supported by the National Basic Research Program of China under Grant 2012CB719904.

The authors are with the School of Remote Sensing and Information Engineering, Wuhan University, Wuhan 430079, China (e-mail: huxy@whu.edu.cn; lixiaokai8990@gmail.com; zhangyj@whu.edu.cn).

Digital Object Identifier 10.1109/LGRS.2012.2205130

The filtering of LiDAR point cloud is an important step in LiDAR data processing. It classifies the LiDAR points into ground points and nonground points, which are objects such as buildings, trees, and low vegetation. Filtering is the first and most important step in producing the digital elevation model (DEM) and terrain information.

Many filtering algorithms have been developed for automatically extracting ground points from the point cloud [2], [3]. These can be divided into several categories, including methods based on mathematical morphology [4]–[7], linear prediction [8], [9], progressive triangulated irregular network (TIN) [10], [11], and segmentation [12], [13]. Sithole and Vosselman [2] tested several algorithms and determined that none could process every type of terrain well. Many researchers are still developing algorithms for the automatic extraction of ground points. For example, Costantino and Angelini [14] and Crosilla *et al.* [15] introduce high-order moments into the classification of point clouds. Currently, researchers are also paying attention to processing efficiency due to the huge amount of data involved in these techniques. Shan and Sampath [16] propose a 1-D approach that conducts the 1-D labeling in two opposite directions, followed by a linear regression. Han *et al.* [17] suggest a new 1-D segmentation algorithm that directly classifies the points into homogeneous groups along a scan line. One-dimensional algorithms are more efficient than existing algorithms, which are mostly based on 2-D neighborhoods.

Parallel processing methods have also been introduced to speed up computation. Han *et al.* [18] use a PC cluster and a virtual grid to create a raster DSM and then produce a digital terrain model using enormous amounts of airborne laser scanning data. Beutel *et al.* [1] construct a grid DEM from massive point clouds using natural neighbor interpolation and get a $10\times$ increase in speed using a graphics processing unit (GPU). Field-programmable gate array (FPGA) [19] and cloud computing [20] are also used to improve processing efficiency. GPU constitutes the most common and cheapest parallel technology. It is relatively difficult to use cloud computing. FPGA is highly customizable and can be very fast, but the developing time and price are high and it is not really suitable for regular computer programmers [21]. Few people have access to a PC cluster. Today, almost all PCs are equipped with a GPU. GPU processing power is growing, and its price is decreasing. After NVIDIA released its Compute Unified Device Architecture (CUDA) [22], GPUs have become more flexible and programmable.

In this letter, we propose a scan line segmentation (SLS) algorithm. First, the local lowest points are detected in a window

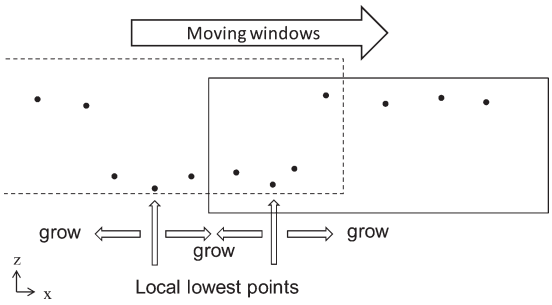

Fig. 1. Diagram of SLS algorithm.

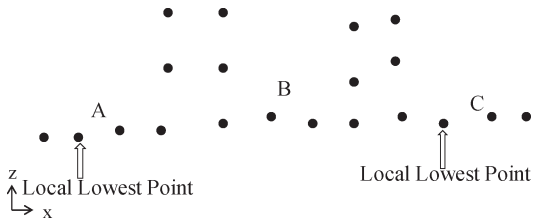

Fig. 2. Case where the method fails.

moving along the scan line. These points are then treated as seeds to grow into ground segments. The algorithm is simple and very efficient. Each scan line is processed independently, so it is suitable for parallel computing. We use a GPU to speed up the SLS algorithm. Experimental results and comparisons with other commonly used methods are presented to evaluate the performance with regard to the filtering error rate and algorithm speed.

II. Scan-Line-Based Filtering Algorithm

In this section, we describe the algorithm explicitly. Most LiDAR systems produce a general sawtooth pattern of measurements of the ground. This scanning mode results in a series of scan lines. There is a 1-b tag to indicate whether the point is the edge of the scan line in the LAS data format. It is easy to obtain the complete data for every scan line, which makes 1-D filtering easy and natural. Our algorithm is based on slope and elevation, and each scan line is regarded as a single process.

The terrain of urban area is usually continuous and smooth, and the slopes are small, ignoring buildings, trees, and so on. We assume that the outliers have been detected and eliminated in advance. Ground points have the lowest elevations in their neighborhoods. Based on these assumptions, we set a window along the scan line and measure the lowest point in the window. Fig. 1 shows the window moving along the scan line and how a series of local lowest points is obtained. We then treat these points as the seed points for segmentation. Comparing the seed points with their neighbors, if the slopes and elevations between the seed points and the adjacent points are less than the given threshold, their neighbors are marked as ground points or end the process in this direction.

Considering the situation shown in Fig. 2, there is no local lowest point around B. Thus, the points near B will not be labeled as ground points. In this case, the points around A or C are used to fit a line, and the line is used to predict the height of the points around B. If the difference of the predicted height and the real one is less than the threshold, the point is labeled as a ground point. Least squares method or others can be used to fit a line.

The pseudocode of the algorithm is as follows.

Step 1) Read the points' coordinates in a scan line.
Step 2) Move the window with a certain step length, and find out the lowest point in each window.
Step 3) /*grow into ground segments */
for local lowest point p_i
/*grow to the right */
for $j = i \rightarrow$ next lowest point id
calculate $h_{j,j+1}$ and $slope_{j,j+1}$
if (p_j is a ground point and $h_{j,j+1} < t_{\text{height}}$ and $slope_{j,j+1} < t_{\text{slope}}$)
mark p_{j+1} as a ground point
if (p_j is not a ground point and ($h_{j,j+1} < -t_{\text{height}}$ or $slope_{j,j+1} < -t_{\text{slope}}$) and $h_{j+1} - h_{\text{predicted}} < t_{\text{height}}$)
mark p_{j+1} as a ground point
end for j loop
/* grow to the left, the same with upper */
end for loop
Step 4) If there are more scan lines to process, go to step 1; else, exit.

Here, h_i is the Z-coordinate of point p_i, $h_{\text{predicted}}$ denotes the height predicted by fitting a line, t_{height} and t_{slope} refer to the thresholds of height difference and slope, $h_{i,j}$ and $slope_{i,j}$ denote the height difference and slope between point p_i and point p_j, and $slope_{i,j}$ is calculated by using the following formula:

$$slope_{i,j} = \arctan\left(\frac{Z_j - Z_i}{\sqrt{(X_j - X_i)^2 + (Y_j - Y_i)^2}}\right), \quad slope_{i,j} \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]. \quad (1)$$

Generally, the length of the moving window should be greater than the maximum object size. In an urban area, 50–70 m is usually suitable. The slope and height thresholds are affected by the conditions of the data set, such as point density and the slope of the terrain. t_{height} can be set to 1 m, and t_{slope} can take 70°–80° for urban areas.

The algorithm is applied to the raw data directly, not to resampled data, to ensure that it avoids any loss of geometric accuracy. It is based on a single scan line, so there is no need to calculate the complex 2-D neighbor relationship, which makes the implementation of the algorithm more efficient. Furthermore, due to the independence of the scan lines, it is highly suitable for parallel computing.

III. Implementation of OpenMP and CUDA Acceleration

Airborne LiDAR is becoming increasingly popular, but the rapid processing of enormous data sets remains a serious problem. Parallel processing is one solution to this problem [20]. In this letter, we use Open Multi-Processing (OpenMP) and CUDA to accelerate the SLS algorithm. This section describes the two parallel technologies briefly and how to apply them to our algorithm. The comparison of their performances is in the next section.

Now that central processing units (CPUs) have entered the multicore era, four- or eight-core CPUs are common. To take full advantage of CPUs' computing ability, it is necessary to use CPU multithread technology. OpenMP, a kind of shared memory architecture application programming interface, is an example of multithreading [23]. It contains a series of compiler directives, and parallelism can be conveniently implemented using these directives. For example, a loop can be parallelized simply by adding one line of compiler directives into the source code. In our SLS algorithm, we use a *for* loop to process all scan lines, and each line is processed independently as mentioned previously. One only needs to add one line of code (*#pragma omp parallel for*) for the *for* loop to achieve parallelism.

GPUs have evolved into highly parallel multithreaded many-core processors with tremendous computational power and very high memory bandwidth [22]. Since the release of CUDA, it has become increasingly convenient and efficient to use GPUs to speed up applications. CUDA manages threads in a hierarchical structure. Threads are organized into a thread block, and the thread blocks are then organized into a grid. There are several memory types in GPU, such as shared memory and global memory. Threads in a block can communicate through high-speed shared memory, while threads in different blocks can communicate only through low-speed global memory [22]. The function that runs on a GPU is called a kernel.

GPU is particularly suitable for running extremely greedy calculations that run massively parallel with limited memory access and flow control. In fact, the clock speed of a single core of GPU is slower than that of CPU. However, GPU has many more cores. By running a large number of threads in parallel, for example, thousands of threads, the memory access latency is hidden at the same time. The LiDAR data often contain thousands of or more scan lines. The scan lines can be processed independently, and the algorithm proposed is simple. It is very suitable to use GPU parallel computation.

In this section, we explain how to use CUDA to implement the parallel computing of our SLS algorithm. The workflow chart is shown in Fig. 3. N refers to the number of scan lines, and n denotes the number of threads in each thread block. Each scan line is processed by a thread block. The shared memory is used to reduce the influence of the high latency of global memory access.

The main processing steps are as follows.

1) Prepare data. Data such as point coordinates and the marks denoting edge points need to be read from the file and arranged.
2) Transfer data. Copy data from CPU dynamic random access memory (DRAM) to GPU global memory.
3) Launch kernel function. First, find the edge point of each window, then compare the points' heights in each window, and obtain the local lowest point. Finally, calculate the slopes and elevations from the lowest point, and grow along the two directions.
4) Copy data back to CPU DRAM.

In step 2, data transfer from DRAM to GPU memory is very slow. *cudaHostAlloc()* is used to allocate page-locked host memory (DRAM) which can increase the transfer bandwidth. The three coordinates of points are organized as shown in Fig. 4 in order to maximize global memory throughput. x_i, y_i, and z_i refer to the three coordinates of point i, respectively.

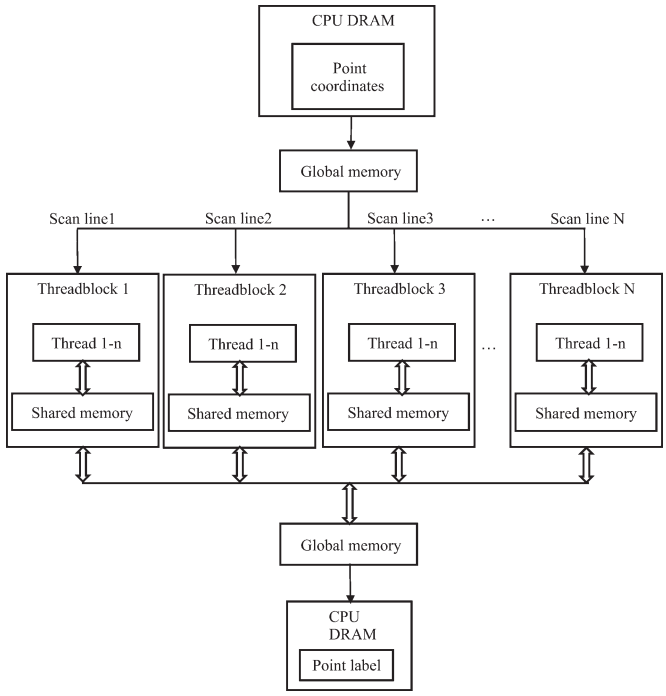

Fig. 3. Workflow of GPU-accelerated filtering.

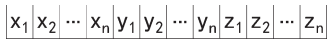

Fig. 4. Organization of point coordinates.

Step 3 can be divided into three substeps.

Step 3.1) Calculate the horizontal distance from each point to the first point in the scan line. Each point is processed by one thread, with the pseudocode shown hereinafter. *BlockSize* refers to the number of threads in every thread block, which is usually set to 128, 256, or 512. The distances (*Dist[]*) are stored in shared memory. Then, use one thread to find the edge of every moving window according to the size of the window, and store the labels of edge points in global memory in order to save the shared memory.

```
Step 3.1
for (i = threadIdx.x; i < pointNum; i+ = BlockSize)
{
Dist[i] = distance from point i to point 0;
}
```

Step 3.2) Read the edge points of each window, and find the local lowest points. One window is processed by one thread. *lowestPoint[]* is stored in shared memory. See the pseudocode hereinafter.

```
Step 3.2
for (i = threadIdx.x; i < windowNum; i+ = BlockSize)
```

```
{
lowestPoint[i] = startPoint;
for startPoint → endPoint
compare the height and update lowestPoint[];
end for
}
```

Step 3.3) Both directions of all lowest points are processed by different threads. It means that two directions from one lowest point are processed by different threads. See the pseudocode hereinafter. Set the value of point's label in global memory to one if its slope and elevation are below a certain threshold.

```
Step 3.3
for (i = threadIdx.x; i < lowestPointNum * 2; i+ = BlockSize)
{
for startPoint → the other lowest points in one direction
compare the slope and elevation with its neighbor and update the label of current point;
end for
}
```

IV. Experimental Results

The proposed algorithm is applied to two real urban data sets. The first data set comes from the city of Huiyan in Guangdong Province in China and contains 249 384 points with a point density of 1.4 points/m^2. It is used to test the error rate of the SLS algorithm. The second data set, which covers the city of Foshan, also in Guangdong Province, has approximately 48 000 000 points with a point density of 0.94 points/m^2. This data set is used to test the performance of the parallel processing.

1) Error Rate of the Algorithm: The Huiyan data contain the typical object types of modern cities, such as roads, high and low buildings, and trees. The terrain is not flat, and the elevation changes over the whole area.

Two types of errors are evaluated: type I (rejection of bare-earth points) and type II (acceptance of object points as bare earth) errors [2]. These rates are then compared with those of the existing algorithms. Table I compares the error rate of our method with the classical filtering algorithms: progressive TIN, linear prediction, and Shan's bidirectional labeling algorithm [2]. A 1-m height threshold and an 80° slope threshold are chosen for the experiment, and the size of the moving window is set to 70 m. The test data are carefully classified by manual editing using TerraSolid's TerraScan to establish a reference. We then use the four algorithms to filter and count the number of ground points.

From Table I, we can see that the progressive-TIN-based algorithm is the best overall. The total error rate of our SLS algorithm is 8.24%. It is a little worse than that of the linear-prediction-based method but better than that of bidirectional labeling algorithm, demonstrating that the proposed SLS method is effective in urban areas. Compared with 2-D filtering algorithms, 1-D algorithms have higher error rates due to their minimal consideration of neighborhoods. However, they are more efficient and faster even without any acceleration. In the SLS algorithm, every local lowest point is used to grow into segments independently. In the bidirectional labeling method, in contrast, each point's label depends on that of the previous point. If the previous point is not labeled correctly due to a complex surface, then all remaining points may be incorrectly processed. Hence, the SLS algorithm is more robust than the bidirectional labeling algorithm, and it is also the fastest.

TABLE I
Quantitative Comparison of Three Algorithms

Algorithm	Type I errors (%)	Type II errors (%)	Total errors (%)	Running time (s)
Progressive TIN	3.74	0.16	3.90	2.25
Linear prediction	6.01	0.23	6.24	7.45
Bidirectional labeling	14.15	1.77	15.92	0.05
SLS algorithm	7.72	0.52	8.24	0.03

2) Computational Performance of Parallel Computing Using CPU Multiple Threads and GPU: In this experiment, the original data set is divided into different sized subsets. Table II shows the running time and the speedup ratio of the SLS algorithm using a single thread, multiple threads, or a GPU to process different amounts of data. The time listed in Table II is the average obtained by running the program 100 times. Fig. 5 shows the trend of the speedup ratio. All experiments are done on a PC with Intel Core i7-920 at 2.67-Hz CPU (four cores and eight logical processors) with 4.0-GB memory, a NVIDIA GeForce GTX285 GPU with 1.0-GB memory, and Windows 7 Ultimate 64-b system.

Table II and Fig. 5 show that a GPU can speed up the computation significantly and is much more powerful than a multicore CPU. In Fig. 5, we can see that, when the number of points is less than 10 million, the speedup ratio increases quickly because data transmission accounts for a smaller and smaller portion of the processing as the amount of data increases. When the number of points is larger than 10 million, the speedup ratio tends to be stable.

V. Conclusion

In this letter, we have proposed a new algorithm that uses scan lines from the LiDAR point cloud to classify the ground and nonground points, and GPU is used to speed up the process. By growing local lowest points into smooth segments, the ground points are obtained. The algorithm by nature is suitable for parallel computing. The test results show that the proposed algorithm is fast and effective in urban areas. The GPU acceleration achieves high performance in computational time. It has the potential to develop into real-time or onboard processing of LiDAR point cloud data during the data acquisition flight.

Due to steeper slopes and dense trees, our algorithm may not work well in mountainous areas. Further research is necessary to extend the method to other terrain types, which would require

TABLE II
RUNNING TIME AND SPEEDUP RATIO

Num. of points (million points)	CPU single thread		Multiple threads by openMP			GPU by CUDA		
	Running time (ms)	Time RMSE (ms)	Running time (ms)	Time RMSE (ms)	Speedup ratio	Running time (ms)	Time RMSE (ms)	Speedup ratio
1	110	5.8	23	7.8	4.71	18	0.8	5.97
5	565	8.1	123	14.5	4.61	67	1.7	8.43
10	1157	9.2	258	22.0	4.49	127	2.0	9.10
16	1810	11.3	403	32.8	4.49	194	2.2	9.33
24	2644	15.1	583	19.4	4.54	288	3.1	9.18
32	3510	16.7	758	24.1	4.63	381	4.7	9.21
40	4385	21.7	928	17.5	4.73	474	5.9	9.26
48	5271	20.6	1104	18.7	4.77	565	6.1	9.34

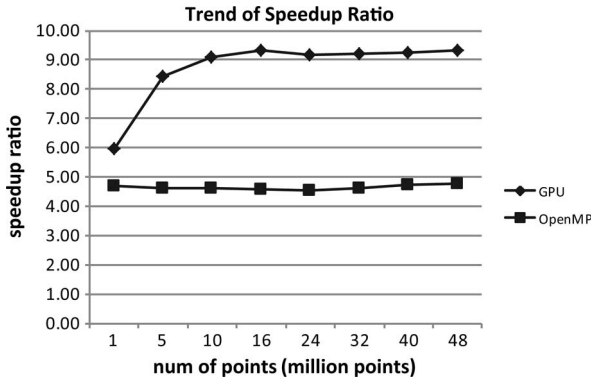

Fig. 5. Trend of speedup ratio.

more complex analysis of scan lines. Moreover, it is clear that parallel computing, accelerated by a GPU, can be used to speed up the process as long as the algorithm is based on scan lines. Progressive-TIN-based filtering algorithm is one of the most commonly used and effective methods. The TIN has to be updated when one point is added to it, so it is not suitable for parallel computation. Considering that one point only affects a few triangles nearby, it is also possible to use GPU to speed up the filtering. This is also one of our research aspects in the future.

ACKNOWLEDGMENT

The authors would like to thank Guangzhou Jiantong Surveying and Mapping Technology Development Ltd. for providing the data for this research.

REFERENCES

[1] A. Beutel, T. Mlhave, and P. K. Agarwal, "Natural neighbor interpolation based grid DEM construction using a GPU," in *Proc. 18th SIGSPATIAL Int. Conf. Adv. Geograph. Inf. Syst.*, New York, 2010, pp. 172–181.

[2] G. Sithole and G. Vosselman, "Experimental comparison of filter algorithms for bare-Earth extraction from airborne laser scanning point clouds," *ISPRS J. Photogramm. Remote Sens.*, vol. 59, no. 1/2, pp. 85–101, 2004.

[3] J. L. S. Cárdenas and L. Wang, "A multi-resolution approach for filtering LiDAR altimetry data," *ISPRS J. Photogramm. Remote Sens.*, vol. 61, no. 1, pp. 11–22, Oct. 2006.

[4] G. Vosselman, "Slope based filtering of laser altimetry data," in *Proc. 33rd Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Amsterdam, The Netherlands, 2000, pp. 935–942.

[5] S. Li, H. Sun, and L. Yan, "A filtering method for generating DTM based on multi-scale mathematic morphology," in *Proc. IEEE Int. Conf. Mechatron. Autom.*, 2011, pp. 693–697.

[6] Q. Chen, P. Gong, D. Baldocchi, and G. Xin, "Filtering airborne laser scanning data with morphological methods," *Photogramm. Eng. Remote Sens.*, vol. 73, no. 2, pp. 175–185, Feb. 2007.

[7] K. Q. Zhang, S. C. Chen, D. Whitman, M. L. Shyu, J. Yan, and C. Zhang, "A progressive morphological filter for removing nonground measurements from airborne LiDAR data," *IEEE Trans. Geosci. Remote Sens.*, vol. 41, no. 4, pp. 872–882, Apr. 2003.

[8] K. Kraus and N. Pfeifer, "Determination of terrain models in wooded areas with airborne laser scanner data," *ISPRS J. Photogramm. Remote Sens.*, vol. 53, no. 4, pp. 193–203, Aug. 1998.

[9] H. S. Lee and N. H. Younan, "DTM extraction of LiDAR returns via adaptive processing," *IEEE Trans. Geosci. Remote Sens.*, vol. 41, no. 9, pp. 2063–2069, Sep. 2003.

[10] P. Axelsson, "DEM generation from laser scanner data using adaptive TIN models," in *Proc. 33rd Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Amsterdam, The Netherlands, 2000, pp. 110–117.

[11] G. Sohn and I. Dowman, "Terrain surface reconstruction by the use of tetrahedron model with the MDL criterion," in *Proc. 34th Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Amsterdam, The Netherlands, 2002, pp. 336–344.

[12] T. Rabbania, F. A. van den Heuvel, and G. Vosselman, "Segmentation of point clouds using smoothness constraint," in *Proc. 36th Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Dresden, Germany, 2006, pp. 248–253.

[13] G. Tolt, Å. Persson, J. Landgård, and U. Söderman, "Segmentation and classification of airborne laser scanner data for ground and building detection," in *Proc. SPIE*, 2006, p. 62140C.

[14] D. Costantino and M. G. Angelini, "Features and ground automatic extraction from airborne LiDAR data," in *Proc. 38th Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Calgary, AB, Canada, 2011.

[15] F. Crosilla, D. Macorig, I. Sebastianutti, and D. Visintini, "Points classification by a sequential higher-order moments statistical analysis of Lidar data," in *Proc. 38th Int. Arch. Photogramm., Remote Sens. Spatial Inf. Sci.*, Calgary, AB, Canada, 2011.

[16] J. Shan and Sampath, "Urban DEM generation from raw LiDAR data: A labeling algorithm and its performance," *Photogramm. Eng. Remote Sens.*, vol. 71, no. 2, pp. 217–226, Feb. 2005.

[17] S. H. Han, J. H. Lee, and K. Y. Yu, "An approach for segmentation of airborne laser point clouds utilizing scan-line characteristics," *ETRI J.*, vol. 29, no. 5, pp. 641–648, Oct. 2007.

[18] S. H. Han, J. Heo, H. G. Sohn, and K. Y. Yu, "Parallel processing method for airborne laser scanning data using a PC cluster and a virtual grid," *Sensors*, vol. 9, no. 4, pp. 2555–2573, 2009.

[19] K. Shih, A. Balachandran, K. Nagarajan, B. Holland, C. Slatton, and A. George, "Fast real-time lidar processing on FPGAs," in *Proc. Conf. ERSA*, Las Vegas, NV, 2008, pp. 231–237.

[20] R. Sugumaran, D. Oryspayev, and P. Gray, "GPU-based cloud performance for LiDAR data processing," in *Proc. 2nd Int. Conf. Comput. Geospatial Res. Appl.*, New York, 2011, p. 48.

[21] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D. J. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *Proc. 16th Annu. IEEE Symp. Field-Programm. Cust. Comput. Mach.*, 2008, pp. 173–182.

[22] *NVIDIA CUDA C Programming Guide*, NVIDIA, Santa Clara, CA, Apr. 6, 2011. [Online]. Available: http://nvidia.com/cuda

[23] OpenMP Architecture Review Board, OpenMP Application Program Interface, Jul. 2011. [Online]. Available: http://openmp.org/wp/about-openmp/