A Streaming Framework for Seamless Building Reconstruction from Large-Scale Aerial LiDAR Data

Qian-Yi Zhou
University of Southern California
qianyizh@usc.edu

Ulrich Neumann
University of Southern California
uneumann@graphics.usc.edu

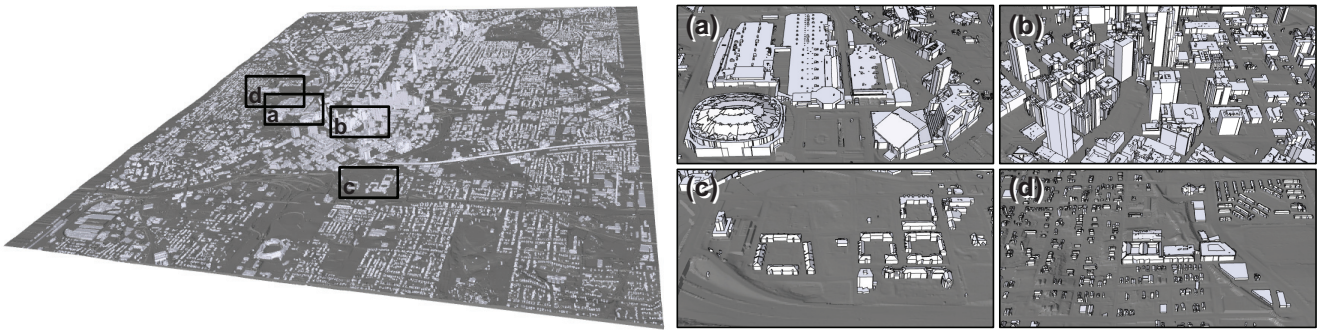

Figure 1. We construct the whole urban model of Atlanta from 683M LiDAR points in a seamless manner, using our streaming building reconstruction program. Right: four closeups of the urban model, showing (a) an area with large flat structures; (b) a downtown area; and (c,d) two residential areas.

Abstract

We present a streaming framework for seamless building reconstruction from huge aerial LiDAR point sets. By storing data as stream files on hard disk and using main memory as only a temporary storage for ongoing computation, we achieve efficient out-of-core data management. This gives us the ability to handle data sets with hundreds of millions of points in a uniform manner. By adapting a building modeling pipeline into our streaming framework, we create the whole urban model of Atlanta from 17.7GB LiDAR data with 683M points in under 25 hours using less than 1GB memory. To integrate this complex modeling pipeline with our streaming framework, we develop a state propagation mechanism, and extend current reconstruction algorithms to handle the large scale of data.

1. Introduction

With the fast development of remote sensing instruments, large LiDAR (light detection and ranging) data sets have become much easier to acquire, and thus are more commonly used to generate building models in urban areas. These building models are very useful for a broad variety of applications such as urban planning, virtual city tourism, disaster simulation, and computer games. Although many research efforts have addressed the building reconstruction problem, most of these methods need to load all the LiDAR data into the memory before processing. Therefore, there is a conflict between the increasing size of data sets and the limitation of computer hardware.

A common way to alleviate this problem is to partition the whole data set into small tiles and process them one at a time. By merging building models generated from different tiles, this approach can produce 3D models from large LiDAR data sets. However, it may introduce artifacts alongside the boundaries between tiles. Although these boundary effects can be moderated by introducing extra processing on boundary regions, the additional processing for tile partitioning, boundary handling and modeling merging is inefficient, tedious and may introduce ambiguity (*e.g.* large buildings that span the intersections of multiple tiles).

We present a framework to handle extremely large data sets in a *seamless* manner, meaning that our method needs no special treatment for tile-boundaries. Our general approach is to adapt existing building reconstruction algorithms to an out-of-core execution architecture. The out-of-core architecture benefits from a *streaming* process which utilizes free hard-disk space as main storage while allocating main memory only as temporary space to store data for ongoing computation. In other words, our *streaming* process takes data as a *stream* (usually, a disk file) - each pipeline component reads from an input stream, loads nec-

978-1-4244-3991-1/09/$25.00 ©2009 IEEE

essary data in-core, processes it; and once data is no longer needed for further processing, it is written into an output stream.

We demonstrate our streaming building reconstruction process by automatically extracting the entire urban models of three different cities. The largest data set we have processed is Atlanta LiDAR data which contains 683M points stored in a 17.7GB disk file. Our program generates 1.12M triangles to represent the buildings and 8.78M triangles for terrain, in under 25 hours of unattended processing using less than 1GB memory. As a comparison, an in-core program would need more than 100GB memory to process this data in one pass. Our resulting Atlanta urban model is shown in Figure 1.

We developed our streaming architecture to support the automatic building reconstruction pipeline described in [17]. This pipeline has the property that most of the operations have *spatial locality*, *i.e*. most computations only require data access within a small local area and are insensitive to global variables (*e.g*. average ground height).

Contributions: While streaming techniques have been widely used in computer graphics, to the best of our knowledge, we are the first to employ them for building reconstruction from aerial LiDAR data. Comparing to the state-of-the-art, we present three key contributions:

1. We present a general streaming framework for processing large-scale LiDAR data. Streaming operators and states are defined formally and a state propagation algorithm is developed to perform streaming operators in a spatial consistent manner.
2. We show how to adapt a building reconstruction pipeline into this streaming framework. As a result, the new pipeline is able to construct urban models from hundreds of millions of LiDAR points in a seamless manner, using a consumer-level PC.
3. We propose novel segmentation and modeling modules to fit our streaming framework. A streaming union-find set partition algorithm is designed for segmentation and a principal direction grid handles the principal direction variation among local regions in large urban cities.

2. Related Work

We review the related work from two aspects: building reconstruction algorithms and streaming approaches.

2.1. Building reconstruction from LiDAR

Pioneer building modeling methods [2, 13, 16] start by converting LiDAR point cloud into a DEM (Digital Elevation Model), and then apply image processing algorithms on these depth images to detect building footprints, fit parametric models and reconstruct polygons. All of them share a similar building reconstruction pipeline with three major steps: classification, segmentation, and building modeling.

Most existing research work is built upon this pipeline and improves the reconstruction quality by improving individual steps. [14] proposes a roof-topology graph to find complex roof patterns from aerial LiDAR data. [3] creates building models with facade by integrating aerial LiDAR and ground based LiDAR. [10] specializes segmentation for densely built areas. [8] presents a two-stages method which can find optimal configuration of parametric models via a RJMCMC sampler. [17] introduces novel algorithms to each of the three steps.

Although automatic solutions have been provided for various LiDAR data sets, to the best of our knowledge, none of them can process an extremely large LiDAR data set in a seamless manner. Instead, many of them (*e.g*. [10, 17]) partition huge LiDAR data into tiles, process them one at a time, and merge the partial results together to generate the aggregate model of a large scale city area. As mentioned previously, artifacts can occur at tile seams and these require special processing, which is often not addressed.

2.2. Streaming methods

To solve the conflict between extremely large data sets and computer hardware limitation, *streaming methods* are developed in geometry modeling and computer graphics areas. They have succeeded in a board variety of applications, such as mesh processing [5] and compression [4], tetrahedral mesh simplification [15], level sets methods [11], point cloud processing [12], LiDAR data rasterization [7], dynamic processing [9], and delaunay triangulation [6].

Here we highlight [6], [7] and [12]. [6] and [7] reveal the local *spatial coherence* of aerial LiDAR data and propose a grid-based indexing structure and a *spatial finalization* mechanism, which is the basis of our approach. [12] performs a sequence of operations on a data stream. To allow data blocks to be in different states and deal with transitions between states, [12] arranges data points into a FIFO queue by sorting the data along one dimension of the largest extent, which is less efficient and general compared with our state propagation mechanism for solving the same problem.

3. Pipeline Overview

An overview of our streaming pipeline is demonstrated in Figure 2. The input LiDAR data (usually stored in a list of disk files) is sequentially read by a pre-processing module called *Finalizer*, which inserts *finalization tags* (markers that indicate spatial coherence) into the data, and produces a *point stream*.

Given the point stream, we design a general approach to perform certain *streaming operations* in a seamless manner requiring only very small amounts of data to be loaded

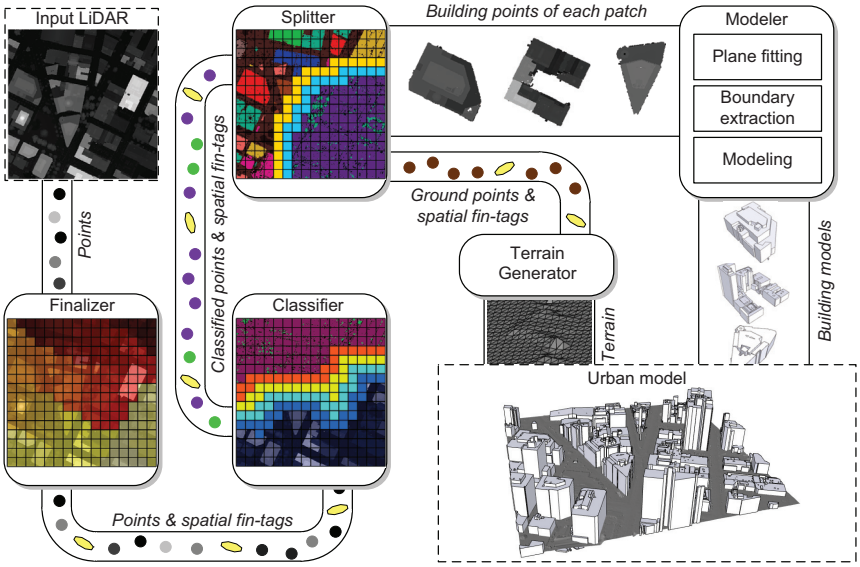

Figure 2. An illustration of the streaming building reconstruction pipeline. A pre-processing module (which is called *Finalizer* in [6]) inserts finalization tags (yellow ovals) and generates a point stream which flows over the *Classifier* and the *Splitter* sequentially. Both components introduce a state-propagation mechanism so that only data with active states (solid colored cells in *Classifier* and *Splitter*) are loaded in-core. The *Splitter* finally generates a point stream and a building stream; which are converted into a terrain model and various building models using the *Terrain Generator* and the *Modeler* respectively.

in-core. Each modeling operation may involve several intermediate steps, which cause data to be in different states. We propose a novel state propagation algorithm to coordinate the state update among partitions of data.

In our pipeline, two specific streaming operations are performed sequentially: the *Classifier* which classifies vegetation points from building and ground points, and the *Splitter* which segments single building patches from the building and ground points. Both are implemented following our formulation of streaming operators and states. In the *Classifier* and *Splitter* blocks of Figure 2, the solid colored cells denote active (*i.e.* in-core) data set with different colors corresponding to different streaming states; the dark red region denotes processed and released data; and the dark blue denotes the input waiting to be read. The active set progresses as a frontier through the input stream until all data is processed.

The *Splitter* outputs two streams: a point stream with geometry information of all ground points and a *building stream* which consists of individual building patches. The point stream is converted to a terrain model using a *Terrain Generator*; and a *Modeler* is responsible for turning each building patch into a polygonal building model. The whole urban model is finally created by combining them together.

4. Streaming Framework

In this section, we define fundamental streaming concepts and present our algorithms for performing general streaming operations.

4.1. Point streams and Finalizer

As observed by [12] and [6], *spatial coherence*, which either appears in the original data set or is the result of a resorting algorithm, can greatly improve the memory efficiency in an out-of-core algorithm. To exploit such spatial coherence, *point stream* is defined as the basic form of data that is processed in a streaming framework:

Definition 1 A *point stream* is a FIFO queue composed of point records and *finalization tags*.

A point stream is generated by inserting *finalization tags* into original input data, which is done by the pre-processing module called *Finalizer*. A finalization tag $f_{\mathcal{A}}$ is a symbol to indicate that all the point records in a spatial area $\mathcal{A}$ has appeared in the point stream before it. This is necessary because the original data usually is not spatially ordered strictly. When a streaming program meets $f_{\mathcal{A}}$, it gets the guarantee that the information within $\mathcal{A}$ is available and further actions can be taken.

We partition the input data into $2^k \times 2^k$ uniform rectangle grid, and take each cell as the basic unit of spatial area in streaming processing. Therefore, the task of our *Finalizer* is to insert one finalization tag for each such cell into the input data.

Note that the finalization tags only provide a mechanism to reveal the spatial coherence but not to generate it. For example, in the worst case, the last point of each grid cell appears at the very end of the input data; the finalization tags will then all be inserted at the end of the stream and no memory efficiency can be produced. Fortunately, the point

sequence in the aerial LiDAR data shows remarkable spatial coherence to make significant memory efficiency [6]. Also, we further enhance the spatial coherence by a *chunking* algorithm which partially resort the point records [6].

We show the finalization result of Oakland data in Figure 6(b). The colors illustrate the time when a grid cell is finalized in the point stream. The spatial coherence between grid cells are revealed by the regularity of the color distribution.

4.2. Streaming operators and states

The basis of our point stream processing framework is the following observation: most of the complicated local algorithms can be decomposed into a series of *streaming operators*, which is a generalized form of the "stream operators" defined in [12].

Definition 2 A *streaming operator* $\Phi_k(\mathbf{p}_i)$ is a local operator which requires all the points in $\mathbf{p}_i$'s local neighborhood $N_k(\mathbf{p}_i)$ to be at *streaming state* s_{k-1} or higher state; $\Phi_k(\mathbf{p}_i)$ takes these points as input and transit point $\mathbf{p}_i$ to *streaming state* s_k.

Here we define a series of *streaming states* $s_0, s_1, \ldots, s_m$. The first state s_0 is always *"Unread"* and the last state s_m is always *"Written and released"*. Except for the last state, the information in a point record at a lower state is always a subset of the information at a higher state. In a complete streaming process, each point sequentially experiences states from s_0 to s_m. And a state transition from s_{k-1} to s_k can only be invoked by a streaming operator Φ_k.

Take our *Classifier* module as an example. All the points are in the initial state s_0 =*"Unread"*. The first streaming operator $\Phi_1(\mathbf{p}_i)$ reads $\mathbf{p}_i$ from the point stream into memory and changes its state from s_0 to s_1 =*"Read"*. The second operator $\Phi_2(\mathbf{p}_i)$ collects the positions of points in $\mathbf{p}_i$'s local neighborhood $N_2(\mathbf{p}_i)$, uses these information to estimate $\mathbf{p}_i$'s normal, then transits $\mathbf{p}_i$ into state s_2 =*"With normals"*. During this process, all the points in $N_2(\mathbf{p}_i)$ must be at least at streaming state s_1, *i.e.* read from the stream and loaded in-core. In a similar manner, the following operators are executed sequentially until point $\mathbf{p}_i$ finally gets into state s_{m-1}. The last stream operator $\Phi_m(\mathbf{p}_i)$ then writes it to the output stream, releases it from memory and turns its state into s_m =*"Written and released"*. In this whole process, point records which are in the first and last states are stored in disk files, and only a small fraction of points in intermediate states need to be loaded in-core. These intermediate states are called *active states*.

To determine when an operator Φ_k can be invoked, we define the *scope radius*:

Definition 3 The *scope radius* $R(\Phi_k)$ is the radius of point $\mathbf{p}_i$'s neighborhood $N_k(\mathbf{p}_i)$ required by $\Phi_k(\mathbf{p}_i)$.

$R(\Phi_k)$ reflects the size of the area affecting Φ_k[1]. In most cases it is determined by the corresponding streaming operator. *E.g.* the *"normal estimation"* operator of the *Classifier* requires a scope radius equal to the neighborhood size δ defined in [17]. The only exception is the scope radius of the last operator Φ_m, which is forced to be the largest scope radius of all the other streaming operators, *i.e.*

$$R(\Phi_m) = \max\{R(\Phi_1), R(\Phi_2), \ldots, R(\Phi_{m-1})\}, \qquad (1)$$

because Φ_m is the only information-subtracting operator – once performed, the point record is no longer available in the memory. By forcing $R(\Phi_m)$ to be the largest scope radius, $\Phi_m(\mathbf{p}_i)$ is applied only when all the points in $N_m(\mathbf{p}_i)$ are at least at state s_{m-1} (written or waiting to be written), so that no point still requires information from $\mathbf{p}_i$ to complete a streaming operator $\Phi_k, k < m$.

The scope radius is also helpful in determining the size of the spatial unit (cell). We require that the side length of a grid cell is no less than $R(\Phi_m)$, so that the impact of any streaming operator applied on a cell c is restricted within its 1-ring neighborhood. This is particularly convenient for the state propagation algorithm in following section.

4.3. State propagation

State propagation is a algorithm which performs the streaming operators in the correct order.

We use cell as the basic unit to perform an operator and denote Φ_k performed on cell $c_{i,j}$ by $\Phi_k(c_{i,j})$. As discussed before, to determine whether $\Phi_k(c_{i,j})$ can be invoked, we only need to check if all points in $c_{i,j}$'s 1-ring neighborhood are at least in state s_{k-1}. In addition, if a cell reaches a new state s_{k-1} by operator Φ_{k-1}, only its 1-ring neighbor cells may receive the direct impact from this transition, *e.g.* a neighbor cell may now satisfy the state prerequisite for Φ_k.

Based on this, the key idea of the state propagation algorithm is to notify all the 1-ring neighbors whenever a cell's state is changed by completing a new operator.

The algorithm starts by reading a cell c_{next} from the input point stream S_{in}. A recursive function *cellAction()* is then called to perform steaming operators in an orderly manner. Taking a cell c and a streaming operator Φ_k as input, *cellAction()* first checks if the state prerequisite for operator $\Phi_k(c)$ is satisfied. If not, it aborts the operation; otherwise, it performs operator $\Phi_k(c)$, transit cell c to state s_k and notifies each cell c^* in c's 1-ring neighborhood by recursively calling *cellAction()* for c^* and operator Φ_{k+1}. In this way, the state change of the initial cell c_{next} is *propagated* in the grid and operators will be performed once they are ready. The pseudo-code of the state propagation algorithm is shown in Table 1.

[1]For convenience, we let $R(\Phi_k) = 0$ when the streaming operator Φ_k does not need any information from the local neighborhood, *e.g.* a *"read from stream"* operator.

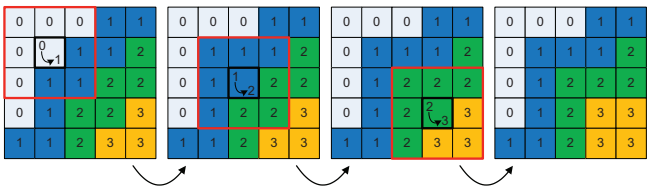

Figure 3. An example of the state propagation algorithm. The numbers and colors denote states. When the state of a cell is changed (marked with black frame), it notifies its 1-ring neighborhood (red frame) to check if any of them is ready for the next operator. The whole process is a recursive procedure.

Figure 3 shows an example of a state propagation procedure in a 4-states problem. The state of each cell is denoted by the number and the color of the cell. In the first step, a cell $c_{1,1}$ (marked by the black frame) is read from the point stream, and its state is transited to s_1 via a call to *cellAction()* function, which then leads a number of streaming operators performed on other cells and state updates. Cells that reach the final state will be written to the output file. The propagation terminates when no more operation is possible, and the state propagation algorithm will read a new cell into the active set and repeat the propagation until all input data is fully processed.

5. Streaming Building Reconstruction

In this section, we show how to adapt a building reconstruction approach similar to [17] into our streaming framework. Modules including streaming classification, streaming segmentation, building modeling and terrain modeling will be described respectively.

5.1. Streaming classification

The classification module is trained by SVM on local geometric features [17] to classify vegetation points from building and ground points. Five types of features based on differential geometry properties are used: regularity $\mathcal{F}_1$, horizontality $\mathcal{F}_2$, flatness $\mathcal{F}_3$, and two normal distribution measurements $\mathcal{F}_4$ and $\mathcal{F}_5$. To compute these features for a point $\mathbf{p}$, CoVariance analysis is performed twice on its local neighborhood. The trained SVM classifier then computes the classification result of $\mathbf{p}$ from these features. Finally, the classification results on $\mathbf{p}$'s neighbor points will vote for the final label of $\mathbf{p}$ in a refinement step.

Since each part of this algorithm requires only the information within a local neighborhood of point $\mathbf{p}$, it is easily decomposed into a series of streaming operators and states shown in Table 2, which are placed into our streaming framework and form up the *Classifier* module.

5.2. Streaming segmentation

The segmentation module aims to divide the input points into a ground patch and individual building patches. [17] adopts a region growing algorithm, but it is difficult to fit into our streaming framework. Hence we propose a novel streaming agglomerative clustering algorithm which is implemented efficiently using the union-find algorithm [1].

The basic agglomerative clustering starts with each point as a segment (or cluster). For each pair of points whose distance is smaller than a certain threshold, the segments they belong to are merged.

The general union-find algorithm utilizes a disjoint-set forests data structure. Hence we use each tree in the forest to store the points in a segment. Each point $\mathbf{p}$ holds a reference $r(\mathbf{p})$ to its parent point in the tree. The segment merging is conveniently implemented by the *union* operation.

To accelerate the process of retrieving root point for each segment, the *find* operation includes a flatten process which links each point on the root-seeking path directly to the root [1]. This flatten process is especially useful in our streaming segmentation; because during the streaming process, some points may be outputted to disk and released from the active set. If it is the parent point of some other points still in the active set, the children's pointers to their parent will become invalid. We resolve this by flattening the tree every time merging is done, and store all the root points in a hash table, so that every active point is guaranteed to point to a parent

/********** **Main program** **********/
Input: a point stream S_{in}; a set of streaming states $\{s_0, \ldots, s_m\}$; and a set of streaming operators $\{\Phi_1, \ldots, \Phi_m\}$.
Output: a point stream S_{out}.
While S_{in} is not empty **do**:
- Read the next cell c_{next} from S_{in};
- Call function ***cellAction***(c_{next},Φ_1).

End of main program.

/********** **cellAction() function** **********/
Input: a grid cell c at state s_{k-1}; and a streaming operator Φ_k.
For each cell c^* in the 1-ring neighborhood of c **do**:
- **if** the state of c^* is lower than s_{k-1}, **then** return *"not ready"*.

/* If not returned, all cells in the 1-ring neighborhood pass the state test. */
Execute $\Phi_k(c)$. /* take action and change the state of c to s_k */
For each cell c^* in the 1-ring neighborhood of c **do**:
- **if** the state of c^* is s_k, **then** call function ***cellAction***(c^*,Φ_{k+1}).

End of cellAction() function.

Table 1. State propagation algorithm.

Streaming operators	Streaming states
Φ_1: Read data from input point stream and allocate memory.	s_0: Unread
Φ_2: Apply CoVariance analysis on positions to estimate normals; and compute $\mathcal{F}_{1,2,3}$.	s_1: Read
Φ_3: Apply CoVariance analysis on normals; calculate $\mathcal{F}_{4,5}$; and apply SVM classifier.	s_2: With normals
Φ_4: Refine classification by making points in local neighborhood vote on result.	s_3: Classified
Φ_5: Write point records to output point stream, and release them from memory.	s_4: Refined
	s_5: Written and released

Table 2. Streaming operators and states for classification.

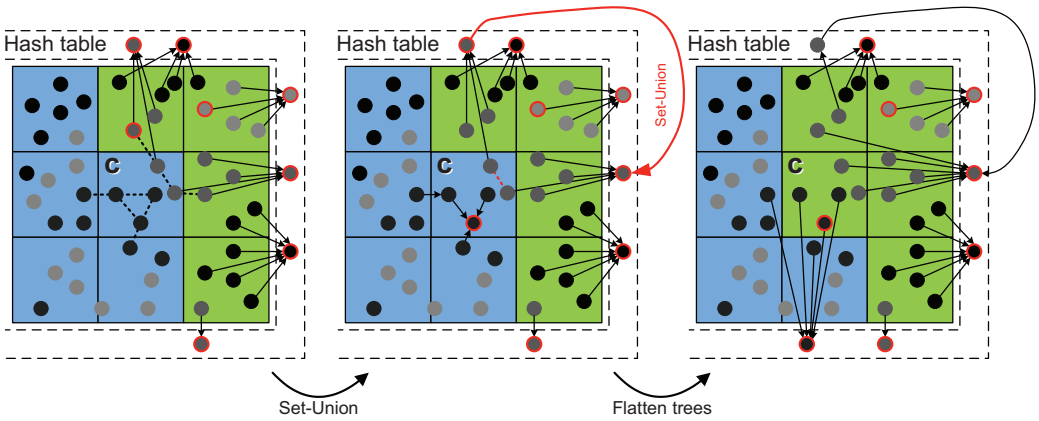

Figure 4. The streaming agglomerative clustering algorithm stores set-tree roots (marked with red frame) in a hash table. It first performs a *union* operation on each neighbor point pair illustrated as the dotted line in the left figure. Then the algorithm flattens the set-trees shown in the middle figure and push new roots into the hash table (right figure).

Input: a cell c at state s_1, with its 1-ring neighborhood at state s_1 or higher; root hash table $\mathcal{H}$; and the distance threshold α.

/* Apply *union* operation over cell c */

For each point pair $(\mathbf{p}, \mathbf{q})$ where $\mathbf{p} \in c$ and $||\mathbf{p} - \mathbf{q}|| < \alpha$, **do**:

- Call function ***union*(p, q)**.

/* Flatten set trees from all touched points */

For each point $\mathbf{p}$ in the 1-ring neighborhood of c **do**:

- Flatten $\mathbf{p}$'s root-seeking path by calling ***find*(p)**.

/* Put new roots into the hash table */

For each point $\mathbf{p}$ in the 1-ring neighborhood of c **do**:

- **if** $\mathbf{p}$ is root and $\mathbf{p} \notin \mathcal{H}$, **then** push $\mathbf{p}$ into $\mathcal{H}$.

End of union-find algorithm.

Table 3. Streaming union-find algorithm (Φ_2).

Streaming operators	Streaming states
Φ_1: Read data from input point stream and allocate memory.	s_0: Unread
Φ_2: Apply streaming union-find algorithm described in Table 3.	s_1: Read
Φ_3: Write point records to output point stream, and release them from memory.	s_2: Segmented
	s_3: Written and released

Table 4. Streaming operators and states for segmentation.

point in the hash table, which will not be released.

The pseudo-code of this algorithm is shown in Table 3. We illustrate the agglomerative clustering process using an example in Figure 4. The algorithm starts with a cell c whose 1-ring neighborhood are all available in memory. The pairs of points involved in c whose distance are small enough for a merging operation is connected in dashed lines. To process c, a *union* operation is performed on each such pair of points and their segments are merged (middle figure). The algorithm then performs a *find* operation over all the points touched in the first step to flatten all the sets which have been changed. Finally, the new roots generated during this process are added into the hash table.

Since this segmentation method is local, it can now be defined as streaming operators. A list of the streaming operators and corresponding states are given in Table 4.

As a result, the output point stream is decomposed into segments of points. The largest segment is taken as the ground patch and sent to the *Ground Generator* still in the form of a point stream. The rest segments are sent to the building modeling module.

5.3. Building modeling

The building modeling algorithm now takes over these building patches. Since the number of points contained in a single building patch is small, the patches are loaded into the memory and processed one by one. In our experiments, the largest building patch is the large structure shown in Figure 1(a), containing 3.2M points, which takes 332MB of memory to process.

We extend the automatic building modeling algorithm in [17] for building model reconstruction. Given a building patch and a set of *principal directions* as input, The algorithm of [17] fits planes to the points and snaps the plane boundary segments onto the principal directions. In [17], the principal directions are extracted over the whole input data, which is problematic since we are dealing with very large-scale input. To allow the principal directions to reflect different boundary directions within local regions (such as the downtown area of Atlanta shown in Figure 1(b) and the residential area shown in Figure 1(d)), we compute a *principal direction grid* (Figure 5). For each cell in this grid, a histogram of the tangent directions of all boundary points is computed within a local neighborhood and the peaks after Gaussian filtering are found to be principal directions.

5.4. Terrain modeling

The objective of the terrain modeling algorithm is to rasterise the ground point stream into a digital elevation model. Taking the point stream as input, the algorithm builds up a

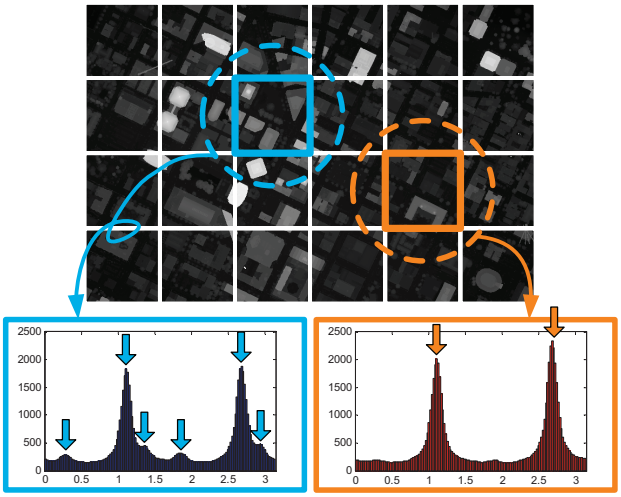

Figure 5. With the principal direction grid on Oakland data set, six principal directions are detected for the blue cell (left), while two principal directions are detected for the orange cell (right).

square grid (whose user-selected unit length determines the precision of the terrain mesh); and counts the lowest ground point in each grid cell. These points are later accepted as vertices of the rasterised terrain model. The empty cells can be filled by solving a Laplace's equation as proposed in [17]. However, for performance reason, we choose to apply a linear interpolation to the gaps along each column on the grid. We observe trivial differences between results generated by these two methods; however, the improvement on efficiency is remarkable for city-scale data sets.

6. Experiments

We tested our streaming building reconstruction on three different data sets, namely, Oakland, Denver, and Atlanta. The problem scale varies from 16M points to 683M points. Our program generates polygonal urban models for each of them. All the experiments are done on a consumer-level PC (Intel Core2 2.4GHz CPU, with 2GB memory and 100GB free hard disk space). The running time and maximum memory usage are reported in Table 5. Although the average processing speed is affected by the characteristics of data sets, it is faster than 3 minutes per million points on all date sets (versus 8 minutes per million points in [17]).

Benefiting from the streaming framework, the memory footprint during our experiments is kept at a low level. The whole pipeline consumes no more than 1GB memory at any time to process our largest data sets (Atlanta) in one pass. We analyze this memory-saving mechanism by showing the *Classifier* module for Oakland data set in Figure 6. First, the finalization result shown in Figure 6(b) reveals the spatial coherence between streaming grid cells. Second, three snapshots are taken during the streaming classification processing (Figure 6(c,d,e)), showing that only cells at active

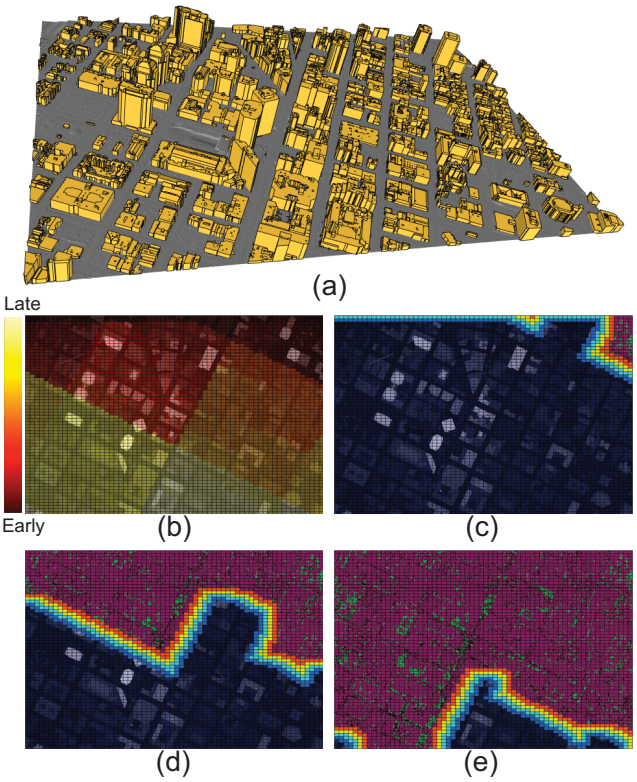

Figure 6. (a) Reconstructed urban model of Oakland downtown area. (b) Finalization result reveals the spatial coherence between cells; colors represent the finalization time. (c,d,e) Three snapshots during the streaming classification algorithm; only a small portion of cells are at active states (bright solid colored cells).

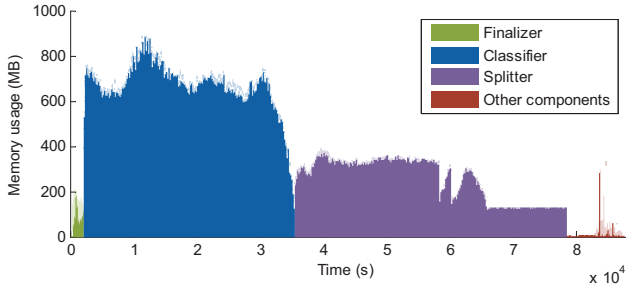

Figure 7. Memory usage during experiment on Atlanta data set.

states (bright solid colored cells) are stored in memory. The remaining cells are either waiting in the input stream or have been written to the output stream and released from memory. With the spatial coherence guaranteed, the active cells are always a small fraction of the data; thus only small amount of memory is required.

We finally demonstrate our results for Atlanta and Denver respectively. The Atlanta urban model is shown in Figure 1, and the memory usage is plotted in Figure 7. The *Classifier* is the most memory consuming module because it has more active states, thus storing more cells in memory; the *Splitter* is the most time consuming module because of

Model	Time (hh:mm:ss)					Maximum memory usage (MB)					Building	Terrain
	Fin.	Cla.	Spl.	Oth.	Total	Fin.	Cla.	Spl.	Oth.	Total	# of tri.	# of tri.
Oakland data set, 1.2km-by-0.8km area, 16M points, one 437MB file, sample rate 17 points/sq.m., grid resolution $64 \times 64 \times 64$												
Oakland	24	11:10	3:14	3:09	17:57	108	276	103	23	276	62K	1.92M
Denver data set, 4km-by-3km area, 73M points, 12 files totally 1.90GB, sample rate 6 points/sq.m., grid resolution $512 \times 512 \times 512$												
Denver	4:43	53:31	2:03:56	15:23	3:17:33	16	156	72	71	156	182K	10.7M
Atlanta data set, 5.5km-by-7.1km area, 683M points, one 11.7GB file, sample rate 17 points/sq.m., grid resolution $512 \times 512 \times 512$												
Atlanta	34:58	9:18:09	11:54:29	2:33:24	24:21:00	209	888	390	332	888	1.12M	8.78M

Table 5. Three data sets with different sample rates are tested using our streaming building reconstruction algorithm on a consumer-level PC. We report the running time and maximum memory usage in each pipeline module, namely, Finalizer, Classifier, Splitter, and other components. The experiment results show the ability of our algorithm to handle extremely large data sets in an efficient manner.

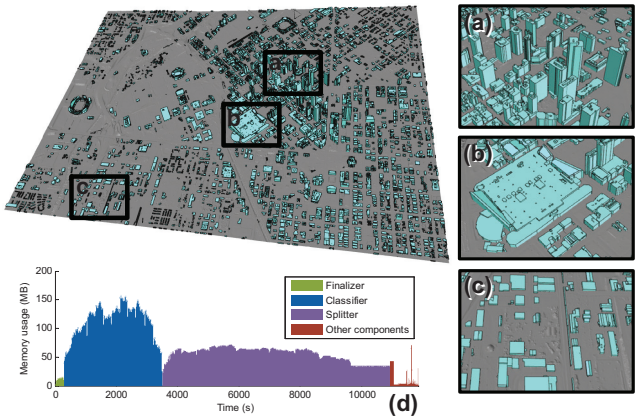

Figure 8. Urban model of Denver with three closeups shown in (a,b,c). Although principal directions in these areas are different; with our principal direction grid, correct principal directions are generated for each of them. (d) Memory usage during processing.

the overhead for saving segmented patches into files. For the Denver data set, its urban model and closeups are shown in Figure 8. Both datasets exhibit variation of principal directions across the whole city. Nevertheless, it is nicely handled by our grid-based principal direction estimation.

7. Conclusions

We present a streaming building reconstruction algorithm which can produce seamless urban models from extremely large aerial LiDAR data sets on a low-end PC. A general streaming framework is proposed with formal definition of streaming operators, streaming states, and a novel state-propagation algorithm for performing streaming operators and state updates in a consistent manner. An automatic building reconstruction pipeline is then adapted into our streaming framework. Experiments are done on several large-scale data sets, which no previous approach is able to process with such a small amount of resource.

Possible future work lies in the following directions. First, our streaming framework is general and extensible to integrate with many other approaches, which may lead to even better performance. Second, when dealing with large-scale data, it is necessary to build more models to describe the differences between local regions. In this work, we focus on the difference of principal directions. Similar idea can be applied to building patterns and vegetation patterns.

8. Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. We gratefully acknowledge the sources of our data sets: Airborne 1 Corp. for Oakland and Atlanta, Sanborn Corp. for Denver. We thank Suya You and Yuan Li for helpful discussion. This work is partially supported by a Provost's Fellowship from USC.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[2] A. Elaksher and J. Bethel. Reconstructing 3d buildings from lidar data. In *ISPRS Commission III, Symposium*, 2002.

[3] C. Früh, S. Jain, and A. Zakhor. Data processing algorithms for generating textured 3d building facade meshes from laser scans and camera images. *IJCV*, 2005.

[4] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM SIGGRAPH*, 2003.

[5] M. Isenburg and P. Lindstrom. Streaming meshes. In *IEEE Visualization*, 2005.

[6] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. In *ACM SIGGRAPH*, 2006.

[7] M. Isenburg, Y. Liu, J. Shewchuk, J. Snoeyink, and T. Thirion. Generating raster dem from mass points via tin streaming. In *Proceedings of Geographic Information Science*, 2006.

[8] F. Lafarge, X. Descombes, J. Zerubia, and M. Pierrot-Deseilligny. Building reconstruction from a single dem. In *CVPR*, 2008.

[9] J.-F. Lalonde, N. Vandapel, and M. Hebert. Data structure for efficient dynamic processing in 3-d. *IJRR*, 2007.

[10] B. Matei, H. Sawhney, S. Samarasekera, J. Kim, and R. Kumar. Building segmentation for densely built urban regions using aerial lidar data. In *CVPR*, 2008.

[11] M. Nielsen, O. Nilsson, A. Soderstrom, and K. Museth. Out-of-core and compressed level set methods. *ACM Transactions on Graphics*, 2007.

[12] R. Pajarola. Stream-processing points. In *IEEE Visualization*, 2005.

[13] F. Rottensteiner. Automatic generation of high-quality building models from lidar data. *IEEE Computer Graphics and Applications*, 2003.

[14] V. Verma, R. Kumar, and S. Hsu. 3d building detection and modeling from aerial lidar data. In *CVPR*, 2006.

[15] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming simplification of tetrahedral meshes. *IEEE Trans. Vis. Comput. Graph.*, 2007.

[16] S. You, J. Hu, U. Neumann, and P. Fox. Urban site modeling from lidar. In *Proceedings, Part III*, ICCSA, 2003.

[17] Q.-Y. Zhou and U. Neumann. Fast and extensible building modeling from airborne lidar data. In *ACM GIS*, 2008.