LASzip: lossless compression of LiDAR data

Martin Isenburg
LAStools
http://laszip.org

***Abstract*—Airborne laser scanning technology (LiDAR) makes it easy to collect large amounts of point data that sample the elevation of the terrain beneath. The LAS format has become the de facto standard for storing and distributing the acquired points. As the sampling density of LiDAR increases so does the size of the resulting files. Typical LAS files contain tens to hundreds of millions points today, but soon billions will be commonplace.**

We describe a completely lossless compression scheme for LiDAR in binary LAS format versions 1.0 to 1.3. Our encoding and decoding speeds are around one to three millions points per second and our compressed files are only 7 to 25 percent of the original file size. Compression and decompression happen on-the-fly in a streaming manner and random-access is supported with a default granularity of 50,000 points. A reference implementation unencumbered by patents or intellectual property concerns is freely available with an LGPL-license, making the proposed compression scheme suitable to become part of the LAS standard.

I. INTRODUCTION

Low flying aircrafts equipped with modern laser-range scanning technology (LiDAR) collect precise elevation information for entire cities, counties, or even states. Shooting 100,000 or more laser pulses per second onto the earth's surface they take measurements at resolutions exceeding one point per square meter. Derivatives of this data such as digital elevation maps are subsequently used in numerous applications: to assess flood hazards, plan solar and wind installations, carry out forest inventories, aid in power grid maintenance, etc. However, the sheer amount of LiDAR data collected poses a significant challenge as not millions but billions of elevation samples need to be stored, processed, and distributed.

The scanner records the waveform of the returning reflection for each laser pulse that it sends out. The intensity peaks of this waveform correspond to points that were hit by the laser and that reflected significant portions back to the sensors on the plane. There can be multiple peaks because the laser may hit several surfaces such as wires or antennas, branches, leaves, or even birds in flight before reaching the ground. Each peak above a certain threshold is called a return. The coordinates of these returns together with intensity, scan angle, GPS time, return number, flight line ID, etc. are the data of interest.

Fresh off the scanner, the LiDAR data is typically stored in a binary, vendor-specific format. But to exchange the data between users and across different software packages it was traditionally converted into a simple ASCII representation where each line was listing the attributes of a single return. While flexible and easy to understand, storing millions (or now billions) of LiDAR returns in a textual format is cumbersome: the file size grows large, parsing the data is inefficient, and it is not possible to seek within the file. Addressing these concerns the ASPRS created a simple binary exchange format - the LAS format [1]. It is now the de facto industry standard for storage and distribution of airborne and mobile LiDAR data.

Up to LAS 1.3, each point record has a core 20 bytes of which 12 bytes store the x, y, and z coordinate as signed integers. The header of a LAS file contains scaling factors for those integers that specify the precision (e.g. such as 0.01 for cm and 0.001 for mm). The other 8 bytes store intensity, scan angle, return count, classifications, etc. This completes the basic point type 0 (for details see Table I). The point types 1 and 3 add an 8 byte GPS time and the point types 2 and 3 provide 6 bytes to store an RGB color. The LAS 1.3 specification introduced the point types 4 and 5 that allow attaching full waveform information to each return (with controversial design choices) but they are not used much.

One of the great features of the LAS format is that it stores the coordinates as scaled and offset integers—thereby requiring the producers to think about the actual precision in their scanned data samples and to choose appropriate increments such as 0.01 meters (or feet) for storing the coordinates. This eliminates the unnecessary if not disastrous bloat of double-precision floats or 20 digit ASCII representations where 15 of the 20 digits are really just scanner noise. The absence of incompressible noise makes it possible to efficiently compress the LiDAR points in a completely lossless manner.

Generic compression schemes are not well suited to compress LiDAR because they do not have the insights into the structure of the data to properly model the probabilities of certain patterns to occur. The WinZIP compressor does not compress well while the WinRAR compressor is extremely slow. Neither scheme is suited for streaming or for random-access decompression, which means the entire file needs to be decompressed before its contents can be accessed.

In this paper we introduce LASzip, a lossless compressor for LiDAR stored in the LAS format. It delivers high compression rates at unmatched speeds and supports streaming, random-access decompression. The source code is available with LGPL-license and was integrated into the open source libraries LASlib of LAStools [2] and libLAS [3]. There is native support for reading and writing LAZ in FME 2012, TopoDOT, VoyagerGIS, and LAStools and others are following. LAZ is used internally at USACE, Certainty3D, Watershed Sciences, Riegl, and others. Data providers such as Open Topography provide LAZ as an compressed download option [4] and the DNR of Minnesota hosts LiDAR for 40 counties exclusively in LAZ format with plans to complete the entire state [5].

II. Background

Before describing the LASzip compressor some preliminaries about coordinate precision, the LAS format, related work in point compression, and entropy and difference coding.

A. Floating-Point Precision vs. Integer Precision

There is a common miss-conception that floating-point representations provide more precision than integer presentations for storing the x, y, z coordinates of a point. They do not.

The coordinates of LiDAR points from an airborne or a mobile survey are spread out in the x-y plane with uniform distribution, in the sense that there are roughly the same number of points per square meter everywhere and that the points are acquired with roughly the same precision. There will not be one particularly dense area where points need to be stored with higher precision. The floating-point format is not designed for storing uniform distributions of numbers.

Storing a number in floating-point representation means that the precision of the number will vary depending on the value of that number. The closer it gets to zero the more precision it will have. This makes it a good format, for example, for numerical computations where more precision is needed closer to zero. But using the floating-point format to store point coordinates means that there is an increasingly precise spacing of data samples around one point—the origin—at the expense of an increasingly imprecise spacing farther away.

An example: in single-precision floating-point there are 2^{23} different numbers to represent a coordinate between 2 and 4 meters with a spacing of $2/2^{23} = 0.00000023841$ meter, there are 2^{23} different numbers to represent a coordinate between 128 and 256 meters with a spacing of $128/2^{23} = 0.00001525$ meter, there are 2^{23} different numbers to represent a coordinate between $524,288$ and $1,048,576$ meters with a spacing of $524,288/2^{23} = 0.0625$ meter, and of course there are also 2^{23} different numbers to represent a coordinate between $2,097,152$ and $4,194,304$ meters with a spacing of $4,194,304/2^{23} = 0.25$ meter. If you notice the pattern you already know that there will also be 2^{23} different numbers to represent a coordinate between $4,194,304$ and $8,388,608$ meters with a spacing of $4,194,304/2^{23} = 0.5$ meter.

In summary, if you store the easting and northing of your coordinates directly in floating-point they may retain just 0.5 meters of precision. If you subtract a constant offset from your coordinates so the origin falls into the middle of the bounding box, then the samples near the origin are stored with incredible precision ... much much more than LiDAR has.

The appropriate format for storing the coordinates of LiDAR points are properly scaled and offset integers. They offer much more *uniform* precision than a corresponding floating-point value for the same number of bits: a 32 bit integer, for example, offers 7 bits more uniform precision than a 32 bit floating-point number [6] and similarly a 64 bit integer offers 10 bits more than a 64 bit float. In order to increase the coordinate range for a large-scale LiDAR collect, the correct thing to do is to move from 32 bit to 64 bit integers. The LAS format [1] uses scaled and offset 32 bit integers.

name of atomic item		**size**	point type and size					
			0	**1**	**2**	**3**	**4**	**5**
point attributes	format	size	20	28	26	34	57	63
POINT10		**20 bytes**	x	x	x	x	x	x
X	int	4 bytes	x	x	x	x	x	x
Y	int	4 bytes	x	x	x	x	x	x
Z	int	4 bytes	x	x	x	x	x	x
Intensity	u_short	2 bytes	x	x	x	x	x	x
Return Number	3 bits	3 bits	x	x	x	x	x	x
Number of Returns of Pulse	3 bits	3 bits	x	x	x	x	x	x
Scan Direction Flag	1 bit	1 bit	x	x	x	x	x	x
Edge of Flight Line	1 bit	1 bit	x	x	x	x	x	x
Classification	u_char	1 byte	x	x	x	x	x	x
Scan Angle Rank	u_char	1 byte	x	x	x	x	x	x
User Data	u_char	1 byte	x	x	x	x	x	x
Point Source ID	u_short	2 bytes	x	x	x	x	x	x
GPSTIME10		**8 bytes**		x		x	x	x
GPS Time	double	8 bytes		x		x	x	x
RGB12		**6 bytes**			x	x		x
Red	u_short	2 bytes			x	x		x
Green	u_short	2 bytes			x	x		x
Blue	u_short	2 bytes			x	x		x
WAVEPACKET13		**29 bytes**					x	x
Wave Packet Descriptor Index	u_char	1 byte					x	x
Bytes Offset to Waveform Data	u_int64	8 bytes					x	x
Waveform Packet Size in Bytes	u_int	4 bytes					x	x
Return Point Waveform Location	float	4 bytes					x	x
X(t)	float	4 bytes					x	x
Y(t)	float	4 bytes					x	x
Z(t)	float	4 bytes					x	x

TABLE I

LASzip groups the attributes of the point types 0 to 5 of the LAS 1.3 format into four atomic items: POINT10, GPS10, RGB12, and WAVEPACKET13 that are then compressed separately.

B. The LAS format

To facilitate the exchange of LiDAR data between data vendors, users, and different software packages, the ASPRS created LAS as a simple binary exchange format [1]. A LAS file of the 1.0 - 1.3 family consists of a header that can be followed by any number of variable length records before the actual point data begins. The first 227 header bytes define the content of a LAS file: the number of variable length records, the offset to the start of the points, the type and size of each point, the number of points, the offsets and scale factors for the integer point coordinates, and a bounding box that describes the extends in x, y, and z of all points in the file.

In LAS 1.3, where each point can have an attached waveform, there are 235 header bytes. The extra 8 bytes describe the start of the waveform data. If this field is zero the waveforms are stored in an external WDP file. If this field is non-zero the waveforms are stored inside the LAS file after the point block and the field contains the offset to the start of the waveform data. LASzip does not (yet) support including the waveform data inside the LAZ file but always writes it to an external WDP file instead—at the moment uncompressed. There is, however, an (undocumented) option in place that will compress the waveforms to a more compact WDZ file.

Full waveform LiDAR data in LAS 1.3 format is currently only produced by one vendor. Apparently, the mechanism for waveform storage was quickly added to the LAS standard to meet the needs of one hardware vendor without seeking mutual consensus among all scanner manufacturers first. There is almost no publicly available waveform data stored in the LAS 1.3 format and there are only a few software products that can make use of the waveform data in LAS 1.3 files. Therefore we postpone the details for full waveform compression for LAS 1.3 until it becomes more relevant.

The point types 0 to 5 available in LAS 1.3 and the attributes they are composed of are detailed in Table I. The LASzip compressor views point types as compositions of four different *atomic* items: POINT10, GPSTIME10, RGB12, and WAVEPACKET13 that are compressed separately. For example, a point of type 3 is composed of POINT10 followed by GPSTIME10 and RGB12. Additionally each point may have n so called "extra bytes" at the end, each of which is currently considered as a BYTE item. These "extra bytes" occur when the LAS header specifies a point size larger than required by the respective point type. For example if the point type is 1 and the point size is 32 then there are 4 "extra bytes".

C. Point Compression

The compression of points has been extensively studied in the context of computer graphics where a point set typically is a dense sampling of a three-dimensional object. We distinguish the following qualities in a compression scheme:

- *lossy* versus *lossless*
  Lossy schemes compress the shape the points represent rather than the exact point coordinates by allowing their positions to change slightly as long as they remain faithful to the underlying surface. They are mainly used in visualization-only applications. Lossless schemes compress point coordinates represented with uniform precision as scaled integers (in literature often referred to as "after bounding box quantization" or as "after quantizing to a certain number of bits per coordinate") exactly.
- *progressive* versus *non-progressive* (or *single-rate*)
  Progressive schemes compress the data such that the decoder can immediately display a lower resolution version of the points while detail is added as the decompression progresses. They are mainly used for instant feedback in an interactive visualization setting. Non-progressive schemes have only one rate of resolution and decompress the points at full precision. They are mainly used as an I/O friendly, alternate format of the point data for transmission or storage, or to take load of a file server.
- *streaming* versus *non-streaming*
  Streaming schemes start compressing the points and outputting the compressed file after reading only a fraction of them and vice-versa start decompressing the points and outputting the decompressed file after reading only a fraction of the compressed data. The memory footprint remains tiny in comparison to the data they process. Non-streaming schemes read all points into memory either during compression, during decompression, or both. They usually need to construct temporary data structures that grow with the number of points. Usually, progressive streaming also falls into this category as decompressing the points to full precision requires keeping all previously decompressed coarser points in memory.
- *point-permuting* versus *order-preserving*
  Point-permuting schemes do not preserve the original point ordering in the file. Their compression gains come in large parts from imposing a clever canonical ordering onto the points that results in small residuals. Order-preserving schemes do not re-order the points. They compress more information about each point as they also need to specify one of $n!$ possible point permutations.
- *sequential* versus *random-access*
  Sequential schemes decompress the points in the order they are encoded into the compressed file. Random-access schemes can seek in the compressed file and only decompress a particular part. The granularity of the random access is typically limited to blocks of points.

The LASzip compressor is *lossless*, *non-progressive*, *streaming*, *order-preserving*, and provides *random-access*.

D. Related Work

The seminal geometry compression paper by Deering [7] sparked the development of a number of compression schemes for meshes [8], [9], [10], [11], [12], [13] that can also be thought of as point compression schemes that encode additional information (i.e. the mesh connectivity). Nearly all point compression schemes assume that the original order of the points is meaningless and permute them as they see fit during encoding to maximize the achieved compression. Because LASzip aims at compressing LAS files exactly—without any modification—reordering the points is not an option.

The kd-tree approach of Devillers and Gandoin [10], [14] recursively bisects a quantized bounding box along all three axis always encoding the number of points in one half. The oct-tree approach of Botsch *et al.* [15] recursively entropy codes for all eight child nodes whether they contain points or not with an 8-bit symbol. Peng and Kuo [12] and Schnabel and Klein [16] use prediction schemes to further improve the bit-rates of the oct-tree approach. Spatial subdivision approaches have the drawback that they do not generalize to include attribute data such as a GPS time or an RGB color.

The method of Waschbüsch *et al.* [17] generates a binary tree over the points by pairing close-by points that are replaced by their centroid to form the next coarser level. The method of Gumhold *et al.* [18] incrementally constructs a prediction tree by greedily attaching the next point to the tree such that it has the smallest possible residual and compress the tree topology and the residuals in a streaming fashion. Merry *et al.* [19] present a more elaborate prediction tree variation that uses a globally minimal spanning tree and a set of predictors.

Quite similar to LASzip are the commercially available LizardTech® LiDAR compressor™ [20] and the LASCompression software [21] that implements the method of Mongus and Zalik [22]. Both schemes specifically target LiDAR points stored in the LAS format and compress them losslessly.

By default the LizardTech's LiDAR compressor [20] encodes points in blocks of 4,096, performing a simplified Haar wavelet transform on each array of point attributes individually. Pairs of attribute values are recursively replaced by an average coefficient, which is simply the left value, and its corresponding detail coefficient, which is the right minus the left value. Because high-order bits of detail coefficients tend to be zero they can compressed efficiently bit-plane by bit-plane using arithmetic coding [23]. The 8 byte floating-point GPS time that is part of some point types is compressed using standard DEFLATE. Besides the compressed contents of the LAS file, the resulting MrSID file also stores spatial indexing information to support area-of-interest queries.

The LASCompression software [21] operates very similar to LASzip in the sense that it predicts the attributes of a point from previous points with a set of prediction rules and compresses the corrective deltas with arithmetic coding. In particular, the authors use a clever scheme for predicting the linear dependencies between successive points that correspond to returns from the same pulse by using the already encoded deltas for x to improve the predictions of y and z.

E. Entropy and Difference Coding

An entropy coder turns a sequence of symbols into a compact stream of bits while using knowledge about the (uneven) distribution of symbols to store them more compactly—up to the theoretical optimum. As the symbol distribution is often not known in advance, an adaptive entropy coder initially assumes it to be uniform and learns the actual distribution along the way. When a symbol stream is expected to have different distributions given "context" information available to the compressor, it is beneficial to switch between different contexts while encoding the symbols. The entropy coder used by LASzip is based on a fast implementation of adaptive, context-based arithmetic coding by Amir Said [24].

A difference coder compresses the current value as the difference to a previous value. This is most effective when the distribution of differences has a much tighter spread and therefore a much lower entropy than the distribution of values. The difference coder used by LASzip entropy codes the number k that describe the tightest interval $[-(2^k - 1), +(2^k)]$ that the difference falls into, entropy codes up to 8 of its highest bits as one symbol while switching contexts for different $k <= 8$, and stores any remaining lower bits raw.

III. The LASzip compressor

LASzip does not compress the LAS header or any of the variable length records. It simply copies them unmodified from the LAS to the LAZ file. It however adds 128 to the value of the current point type to prevent standard LAS readers from attempting to read a compressed LAZ file. It also adds one variable length record that specifies the composition of the compressed points and various compression options used.

Tambien cambia el offset, por el VLR que añade

The LASzip compressor views the different point types of the LAS 1.3 specification as compositions of items: POINT10, GPSTIME10, RGB12, WAVEPACKET13, and BYTE. Each item has its own compressor with its own version number making the compressor modular and easy to extend to future point types. This document describes LASzip 2.0 that uses version 2 compression for all items except WAVEPACKET13. The earlier LASzip 1.0 uses version 1 compression exclusively and does not allow random access decompression. While still supported in software for backward compatibility, we do not describe the LASzip 1.0 compressor in this document.

The LASzip compressor always encodes the points in *chunks* of points to allow seeking in the compressed file. The default *chunk size* is currently set to 50,000 points. Chunking makes it possible to, for example, augment the produced LAZ files with spatial indexing LAX files [2] and support area-of-interest queries that decompress only the relevant parts of a compressed LAZ file. Because each compressed chunk is different in size, the compressor stores a *chunk table* at the end of the file that specifies the starting byte of each chunk.

When starting a new chunk, the LASzip compressor stores the first point as raw bytes and then initializes the entropy coder. This point is then used as the initial value for subsequent prediction schemes. All following points are compressed item by item with the compression schemes detailed below.

A. Compressing POINT10 (version 2)

First, the compressor encodes a bit-mask of 6 bits that specifies whether any of the following attributes have changed in comparison to the previously processed point:

- **intensity**: the 2 bytes that specify the unsigned 16 bit intensity value i of the point stored in bytes 12 and 13.
- **return bits**: the 3 + 3 + 1 + 1 bits that specify the return number r, the number of returns of given pulse n, and scan direction and edge of flight line stored in byte 14.
- **classification bits**: the 5 + 1 + 1 + 1 bits that specify the 5-bit ASPRS classification of the point c and the synthetic, keypoint, and withheld flags stored in byte 15.
- **scan angle rank**: the 1 byte that specifies the signed 8-bit scan angle a of the point stored in byte 16.
- **user data**: the 1 byte that specifies the unsigned 8-bit user data u of the point stored in byte 17.
- **point source ID**: the 2 bytes that specify the unsigned 16-bit flight line p of the point stored in bytes 18 and 19.

Next, the compressor encodes all attributes that have changed. It encodes the **return bits** while switching between 256 entropy contexts in dependence on the previous return bits byte. The return number r and the number of returns of given pulse n are used to derive two numbers that are used for switching contexts later: a return map m and a return level l.

The return map m simply serializes the valid combinations of r and n: for $r = 1$ and $n = 1$ it is 0, for $r = 1$ and $n = 2$ it is 1, for $r = 2$ and $n = 2$ it is 2, for $r = 1$ and $n = 3$ it is 3, for $r = 2$ and $n = 3$ it is 4, for $r = 3$ and $n = 3$ it is 5, for $r = 1$ and $n = 4$ it is 6, etc. Unfortunately, some LAS files start numbering r and n at 0, only have return numbers r, or only have number of return of given pulse counts n. We therefore complete the table to also map invalid combinations of r and n to different contexts as shown below.

file name	file size [MB]			compression ratio		enc. time [sec]		dec. time [sec]	
	LAS	LAZ	SID	LAZ	SID	LAZ	SID	LAZ	SID
5126-05-57.las	287	21	68	13.4	4.2	4.1	85	20	62
5126-05-58.las	312	29	80	10.6	3.9	4.7	97	22	72
5126-05-59.las	363	45	104	8.1	3.5	5.9	120	25	100
5126-05-60.las	287	21	68	13.5	4.2	4.1	87	20	58
5126-05-61.las	286	21	68	13.4	4.2	4.1	87	20	58
total	**1,534**	**138**	**388**	**11.1**	**4.0**	**23**	**476**	**106**	**350**
1942-29-59.las	486	83	144	5.9	3.4	18	242	35	156
1942-29-60.las	485	81	142	6.0	3.4	15	246	33	154
1942-29-61.las	480	80	140	6.0	3.4	14	234	33	153
1942-29-62.las	464	77	135	6.0	3.4	13	224	31	143
1958-23-23.las	539	86	156	6.3	3.4	14	268	37	173
total	**2,454**	**407**	**716**	**6.0**	**3.4**	**74**	**1,214**	**169**	**779**

TABLE II
PERFORMANCE COMPARISON BETWEEN LASZIP (LAZ) AND THE LIZARDTECH LIDAR COMPRESSOR (SID) IN COMPRESSION RATIO AND ENCODING/DECODING TIMES FOR LIDAR OF THE MINNESOTA DNR [5].

file name	file size [MB]			compression ratio		enc. time [sec]		dec. time [sec]	
	LAS	LAZ	LCMP	LAZ	LCMP	LAZ	LCMP	LAZ	LCMP
5126-05-57.las	287	21	26	13.4	11.1	21	119	20	125
5126-05-58.las	312	29	38	10.6	8.2	21	161	22	149
5126-05-59.las	363	45	59	8.1	6.1	24	240	25	120
5126-05-60.las	287	21	26	13.5	11.0	19	120	20	65
5126-05-61.las	286	21	26	13.4	11.1	17	122	20	65
total	**1,534**	**138**	**175**	**11.1**	**8.8**	**102**	**762**	**106**	**524**
1942-29-59.las	486	83	94	5.9	5.2	42	212	35	276
1942-29-60.las	485	81	91	6.0	5.3	40	281	33	363
1942-29-61.las	480	80	90	6.0	5.3	37	348	33	355
1942-29-62.las	464	77	87	6.0	5.3	37	343	31	345
1958-23-23.las	539	86	97	6.3	5.5	41	380	37	379
total	**2,454**	**407**	**460**	**6.0**	**5.3**	**198**	**1,564**	**169**	**1,718**

TABLE III
PERFORMANCE COMPARISON BETWEEN LASZIP (LAZ) AND THE LASCOMPRESSION CODER (LCMP) [21] IN COMPRESSION RATIO AND ENCODING/DECODING TIMES FOR LIDAR OF THE MINNESOTA DNR [5].

```
const U8 return_map_m[8][8] =
{
  { 15, 14, 13, 12, 11, 10,  9,  8 },
  { 14,  0,  1,  3,  6, 10, 10,  9 },
  { 13,  1,  2,  4,  7, 11, 11, 10 },
  { 12,  3,  4,  5,  8, 12, 12, 11 },
  { 11,  6,  7,  8,  9, 13, 13, 12 },
  { 10, 10, 11, 12, 13, 14, 14, 13 },
  {  9, 10, 11, 12, 13, 14, 15, 14 },
  {  8,  9, 10, 11, 12, 13, 14, 15 }
};
```

The return level l specifies how many returns there have already been for a given pulse prior to this return. Given only valid combinations for the return number r and the number of returns of given pulse n we could compute it as $l = n - r$. But we again use a completed look-up table as shown below to map invalid combinations for r and l to different contexts.

```
const U8 return_level_l[8][8] =
{
  {  0,  1,  2,  3,  4,  5,  6,  7 },
  {  1,  0,  1,  2,  3,  4,  5,  6 },
  {  2,  1,  0,  1,  2,  3,  4,  5 },
  {  3,  2,  1,  0,  1,  2,  3,  4 },
  {  4,  3,  2,  1,  0,  1,  2,  3 },
  {  5,  4,  3,  2,  1,  0,  1,  2 },
  {  6,  5,  4,  3,  2,  1,  0,  1 },
  {  7,  6,  5,  4,  3,  2,  1,  0 }
};
```

The LASzip compressor then encodes the **intensity** as a difference to the most recent intensity with the same return map m. The intuition behind this is that, on average, a single return (where $r = 1$, $n = 1$, $m = 0$) tends to have a different intensity than the first return of a double return (where $r = 1$, $n = 2$, $m = 1$) or the last return of a triple return (where $r = 3$, $n = 3$, $m = 5$). The compressor also switches between 4 entropy contexts $m = 0$, $m = 1$, $m = 2$, and $m > 3$ to further correlate the expected differences in intensity distributions.

The compressor then encodes the **classification bits** while switching between 256 entropy contexts depending on the previous return classification byte. There is a potential to improve compression further by switching contexts based on the return map m as, for example, a single return is more likely to be classified as “building” or “ground” whereas the first return of many is more likely to be “vegetation” or “wire”. We can expect a modest compression gain from this and plan to implement this for compressing the new point types of the recently released LAS 1.4 specification [1].

The LASzip compressor then encodes the **scan angle rank** as the difference to the previous scan angle rank. It switches between two entropy contexts based on the scan direction flag. Next, LASzip encodes the **user data** while switching between 256 entropy contexts in dependence on the previous user data byte, before it encodes the **point source ID** as the difference to the previous point source ID. Remember that each of these six attributes is only encoded if its value has changed.

Finally the compressor encodes the x, y, and z coordinates. Rather than compressing coordinates directly, LASzip predicts them from previous points and entropy codes the difference. For the x and y coordinate it uses a second order predictor: it predicts the coordinate differences d_x and d_y between the previous and the current point as the median of the five immediately preceding differences of points with the same return map m. The intuition behind this is, for example, that single returns are always from a different laser pulse than the previous point and therefore have a wider spacing in x and/or y than the middle of three returns.

For the z coordinate (the elevation) LASzip uses a first order predictor: it predicts z as the elevation of the immediately preceding point of the same return level l. The intuition is, for example, that in a forested area a higher return level l signals a deeper penetration into the forest canopy and therefore a lower elevation. However, the first of a double return hitting a power-line has the same return level as a single return hitting the ground. We can get a small compression gain from using the return map m instead of the return map l. We plan to implement this for compressing the new point types of the recently released LAS 1.4 specification [1].

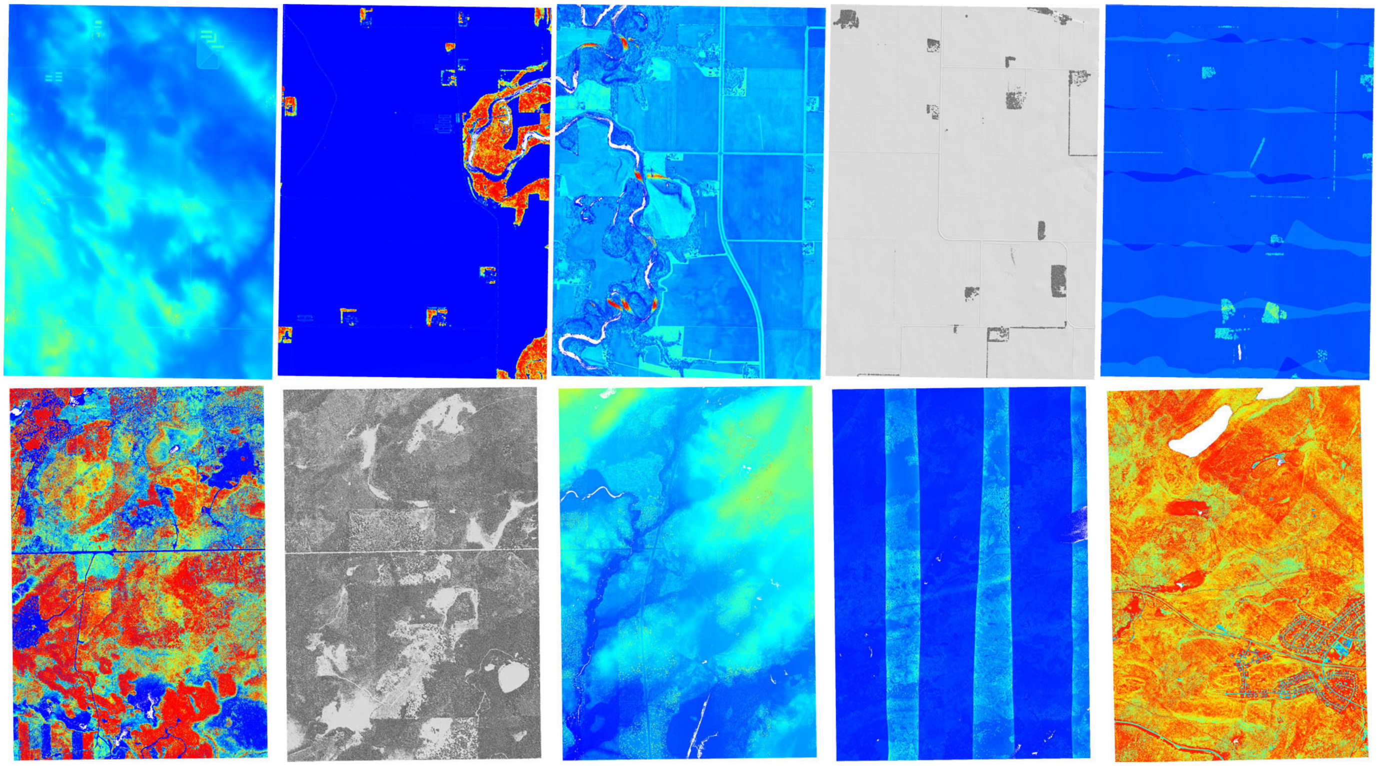


Fig. 1. The data-sets used in Tables II, III, and IV are provided by the DNR Minnesota [5]. Shown are various derivatives such as false-color elevation, standard deviation of elevation, highest intensity, hill-shaded elevation, and point densities generated with `lasgrid` and `blast2dem` from LAStools [2].

B. Compressing GPSTIME10 (version 2)

The GPS times of a single flight path are a monotonically increasing sequence of double-precision floating-point numbers where returns of the same pulse have the same GPS time and where subsequent pulses have a more or less constant spacing in time. While the LASzip compressor is optimized for compressing single flight paths it will handle any GPS time sequence. The compression ratio depends on how far the input is from the expectations. For randomly permuted points it will be terrible. For multiple flight paths that have been sorted into tiles one after another it will be excellent.

For compression purposes LASzip treats double-precision floating-point GPS times as if they were signed 64 bit integers and predicts the deltas between them. As prediction contexts, it stores up to four previously compressed GPS times with corresponding deltas. Keeping multiple prediction contexts can account for repeated jumps in GPS time that arise when multiple flight paths are merged with fine spatial granularity.

LASzip distinguishes several cases that are entropy coded with 516 symbols depending on if the current GPS time is

0	predicted with a delta of zero.
1–500	predicted using the current delta times 1 to 500.
501–510	predicted using the current delta times -1 to -10.
511	identical to the last.
512	starting a new context.
513–515	predicted with one of the other three contexts.

For the first three cases LASzip subsequently difference codes the delta prediction and the actual delta. Nothing further is coded when the GPS times are identical. LASzip starts a new context when the delta overflows a 32 bit integer. For that it difference codes the 32 higher bits of the current GPS time and the current context and stores the lower 32 bits raw. Otherwise it switches to the specified context (where the delta will not overflows a 32 bit integer) and recurses. The current deltas stored with each context are updated to the actual delta when they were outside the predictable range more than 3 consecutive times (i.e. bigger than 500 times the current delta, smaller than -10 times the current delta, or zero).

Currently the LASzip compressor does not make use of known data from the already compressed item POINT10 for compressing GPSTIME10. However, if return counts and point source IDs are populated correctly there is significant correlation that can be exploited. For example, subsequent returns of the same pulse are likely to have the same exact GPS time, and subsequent returns with different point source IDs are likely to require a context switch. We plan to exploit this when compressing the new point types of the recently released LAS 1.4 specification which include the GPS time as an integral part of the point [1].

C. Compressing RGB12 (version 2)

LAS uses unsigned 16 bit integers for the R, G, and B channel. Some files—incorrectly—populate only the lower 8 bits so that the upper 8 bits are zero. Other files—correctly—multiply 8-bit colors with 256 so that the lower 8 bits are zero. The LASzip compressor therefore compresses the upper and lower byte of each channel separately. First it entropy codes 6 bits that specify which bytes have changed as one symbol. For all bytes that have changed it then entropy codes the difference to the respective previous byte modulo 256.

The channels are encoded in the order R, G, and B. Differences encoded in earlier channels are used to predict differences in later channels as there tends to be a correlation in the intensity across channels—especially for gray colors. For example, if there was a byte difference in the low byte of the R channel that difference is added to low byte of the G channel which—clamped to a 0 to 255 range—becomes the value to which the difference of the current low byte is computed.

D. Compressing WAVEPACKET13 (version 1)

The LASzip compressor supports compression of point types 4 and 5 that contain wave packet information. However, the current scheme is still in version 1 as it has yet to be optimized. So far there has been very little real-world demand for compressing LAS files containing waveform data simply due to a lack of data stored in this format.

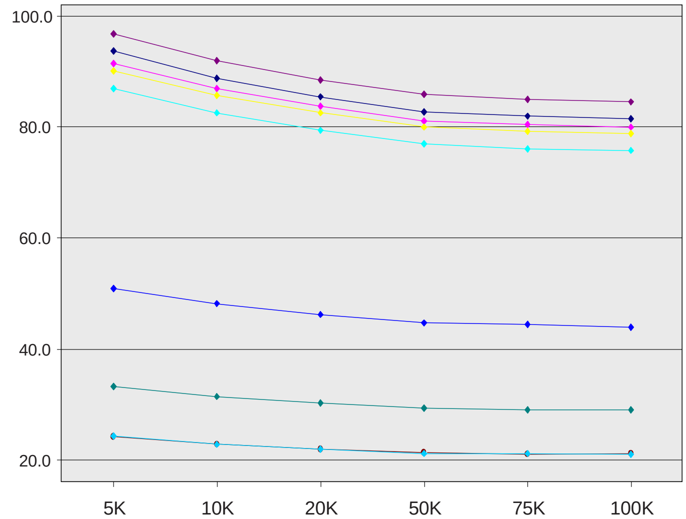
LASzip simply entropy codes the **wave packet descriptor index**, an unsigned byte that is zero if a point has no waveform and indexes the variable length record describing the format of the waveform otherwise. To compress the **bytes offset to waveform data** it entropy encodes one of 4 possible cases:

1) same as last offset
2) use last offset plus last packet size
3) difference to last offset is less than 32 bits
4) difference to last offset is more than 32 bits

In the first two cases no other information is needed. For the other two cases LASzip difference codes the 32 or the 64 bit numbers. The LASzip compressor difference coded all remaining fields. Only **waveform packet size in bytes** is an integer number. The **return point waveform location**, **x(t)**, **y(t)**, and **z(t)** are single-precision floating-point numbers whose 32 bits are treated as if they were a 32-bit integer.

E. Compressing BYTE (version 2)

A LAS point may have “extra bytes” because the LAS header specifies a point size larger than required by the respective point type. Each “extra byte” is entropy encoded with its own context as the difference to the previous “extra byte” modulo 256. Treating them as individual bytes is currently the best that the LASzip compressor can do as there is no description in the LAS 1.3 specification what these “extra bytes” may mean. Six “extra bytes”, for example, could be a single-precision float storing the echo width followed by an unsigned short storing the normalized reflectivity. Or it could be an unsigned short storing a tile index followed by an unsigned integer storing the original index of the point. The recently released LAS 1.4 specification now officially has an “Extra Bytes” variable length record to describe the structure and the individual data types of “extra bytes” [1], which will allow compressing them more appropriately in the future.



file name	uncompressed size	compressed size [MB] 5 K	10 K	20 K	50 K	75 K	100 K
5126-05-57.las	287	24.3	22.9	22.0	21.3	21.2	21.2
5126-05-58.las	312	33.4	31.5	30.2	29.4	29.1	29.1
5126-05-59.las	363	51.0	48.2	46.2	44.8	44.5	44.1
5126-05-60.las	287	24.4	22.9	22.0	21.3	21.2	21.1
5126-05-61.las	286	24.4	23.0	22.1	21.4	21.2	21.1
total	**1,534**	**157**	**149**	**143**	**138**	**137**	**136**
1942-29-59.las	486	93.7	88.9	85.5	82.7	82.0	81.6
1942-29-60.las	485	91.6	87.0	83.9	81.2	80.5	80.1
1942-29-61.las	480	90.2	85.7	82.6	80.0	79.3	78.9
1942-29-62.las	464	87.0	82.6	79.5	77.1	76.2	75.9
1958-23-23.las	539	96.8	92.0	88.6	85.9	85.1	84.6
total	**2,454**	**459**	**436**	**420**	**407**	**403**	**401**

TABLE IV
THE EFFECT OF DIFFERENT CHUNK SIZES 5,000, 10,000, 20,000, 50,000, 75,000, AND 100,000 POINTS ON THE COMPRESSION RATES OF LASZIP.

IV. RESULTS

All timings were taken on an old (2005) Dell Inspiron D6000 laptop with a 2.13 Ghz Intel processor and an even older (2003) external LaCie 120 GB fire-wire drive. Encode timings include reading the uncompressed LAS file from the local disk (the disk cache was flushed) and writing the compressed LAZ file to the external fire-wire disk. Decode timings include reading the compressed LAZ file from the

LiDAR of Minnesota DNR by county name	# of files	# of points	size [GB] LAS	LAZ	compression ratio
cottonwood	216	2,491,327,766	65.0	5.2	12.6
douglas	252	2,092,702,039	54.6	5.4	10.2
freeborn	260	1,713,544,294	44.7	3.7	12.1
houston	197	1,450,109,156	37.8	3.9	9.8
jackson	240	2,724,531,642	71.0	5.6	12.6
lincoln	195	2,200,533,847	57.4	4.5	12.8
martin	252	2,853,353,232	74.4	5.8	12.9
murray	252	2,872,608,269	74.9	5.8	12.9
pope	247	2,404,624,049	62.7	5.3	11.9
redwood	302	3,505,060,711	91.4	7.3	12.4
sibley	219	2,501,934,963	65.2	5.8	11.2
swift	282	2,931,687,204	76.4	5.8	13.2
total	**2,914**	**29,742,017,172**	**776**	**64**	**12.1**

TABLE V
LASZIP COMPRESSES 30 BILLION POINTS (OR 12 COUNTIES WORTH OF LIDAR) FROM 776 GB OF LAS DOWN TO 64 GB OF LAZ FOR LIDAR HOSTED BY THE DNR OF MINNESOTA AT FTP://LIDAR.DNR.STATE.MN.US.

external fire-wire disk and writing the uncompressed LAS file back to the local disk. Reading and writing from different disks makes the process less I/O bound. Nevertheless, the CPU usage for LASzip averages only 50% to 70% on this laptop. When decompressing LAZ files into memory (not measured here) LASzip is entirely CPU-bound running at 99%.

The most common question about LASzip is how it compares to the LizardTech® LiDAR compressor™ that generates the well-known MrSID format [20]. The results in Table II show a comparison of the two compressors in terms of compression ratio, encoding speed, and decoding speed on typical LiDAR tiles that are publicly provided by the Minnesota Department of Natural Resources [5]. LASzip is a clear winner over MrSID in all three performance measures. The compressed LAZ files are 2 to 3 times smaller than the compressed SID files, compression was 16 to 20 times faster, and decompression was 3 to 4 times faster. The encode (!) times were taken at the Minnesota DNR on a Dell Precision T3400 workstation with a 3.14 Ghz Intel processor. The decode times were taken on the Dell laptop using the command-line `lidardecode.exe` version 1.1.0.2802 [20].

We need to be a bit careful in comparing LASzip and the LizardTech LiDAR compressor because they are aimed at different work-flows: LASzip is designed to turn large LAS files into more compact LAZ files for easier management, faster transmission, and lower file system I/O. The LizardTech product adds extra value by seamlessly integrating into the established raster work-flow of the MrSID file format and includes multi-resolution support for fast access to the point data at global scale with an option for lossy compression.

In Table III we compare the performance of LASzip with the LASCompression software [21] that implements the algorithm described by Mongus and Zalik [22] on the exact same datasets. LASzip consistently achieves between 15 to 25 percent higher compression, encodes 7 to 8 times faster, and decodes 5 to 10 times faster. These are end-to-end wall-clock timings taken under the exact same conditions for I/O performance of the laptop / fire-wire disk configuration. While compression rates are comparable, LASzip clearly excels in speed.

The impact of different chunk sizes on the achieved compression is illustrated in Table IV. Smaller chunk sizes mean less compression as the adaptive entropy coder resets at the start of each chunk and needs to relearn all symbol distributions, which negatively affects compression. There is little to be gained from chunk sizes larger than 50,000 and there is no reason for choosing chunk sizes smaller than 5,000 as LiDAR is usually processed in increments of millions of points.

A large-scale user of LASzip is the Department of Natural Resources of Minnesota that—at the time of writing—hosts 40 counties of publicly accessible LiDAR in LAZ format [5]. An overview of the savings in data storage, transmission bandwidth, and download time for 12 of those counties is detailed in Table V. The 30 billion points that would take up 776 GB if stored as LAS files compress down to 64 GB as LAZ.

Compression performance across a large smorgasbord of typical, experimental, as well as unusual LAS files is reported in Table VI with the number corresponding to the smallest file size (or the highest compression ratio) being in bold. We also report the point type and loosely categorize the point order as both have an effect on the compression rates. Point order f means in flight-line order, point order x means sorted along some axis, and point order t means some form of tiling.

The standard WinZIP algorithm is by far the worst performer. This is noteworthy because WinZIP is still used by several agencies to provide compressed LAS downloads. In contrast, the generic WinRAR algorithm does surprisingly well. While it takes a long time to compress, in terms of compression rate it gives the dedicated LizardTech LiDAR compressor a run for its money. The LT compressor struggles with point types 1 and 3 that contain an 8 byte floating-point GPS time, which it compresses with the inefficient DEFLATE.

Again, LASzip gives the overall best compression ratio. The data sets on which it is outperformed are usually those sorted along an axis (i.e. point order x) or those with little oddities such as "Grass Lake Small", for example, which has random values in its return number field and its classification field or "line_27007_dd", which has a strange z coordinate scaling. Although not reported here, LASzip is across the board the—by far—fastest algorithm for both compression and decompression.

V. Discussion

Compressing LAS with WinZIP, WinRAR, gzip, bzip, or any other generic compressor not only means settling for larger files and slower encoding and decoding speeds, but also means that no seeking is possible in the compressed file and that accessing any part of the LiDAR requires to completely decompress the file first. LASzip allows you to treat compressed LAZ files just like standard LAS files. You can load them directly from compressed form into your application without needing to decompress them onto disk first. The availability of two APIs, LASlib [2] and libLAS [3], with LASzip capability makes it easy to add native LAZ read/write support to your own software package. The LASzip source code is available on the website indicated above.

The LASzip compressor is optimized for the case where points are stored more or less in scanner acquisition order in the LAS file and the compression rates degrades the farther the file is from that assumption. If a LAS file is a tile that is part of a larger tiling, the best compression rates are achieved when the flight-lines that pass through the tile are kept in acquisition order (e.g. like `lastile` from LAStools does it). Some LiDAR processing software disturbs the original point order and produces seemingly meaningless point permutations. When compressing large LiDAR collects to be offered via a web server to a large audience it may make sense to first re-order the points of each tile into acquisition order (e.g. with `lassort` from LAStools).

The compressed LAZ files can - just like the original LAS files - be used in conjunction with the small spatial indexing LAX files (that can be produced with `lasindex`). This supports efficient area-of-interest queries when reading the

original LAS file		point		compression ratio					total file size in MB					
name	size in bytes	type	order	ZIP	RAR	LCMP	SID	LAZ	LAS	ZIP	RAR	LCMP	SID	LAZ
Grass Lake Small	123,876,781	0	x	2.6	6.1	**7.0**	6.9	6.2	118	46	19	**17**	17	19
LASFile_1	48,097,847	0	f	1.9	3.8	**4.8**	4.3	4.8	46	24	12	**10**	11	10
LASFile_2	44,168,907	0	f	1.9	3.8	4.9	4.5	**4.9**	42	22	11	9	9	**9**
LASFile_3	16,782,887	0	f	1.8	3.6	4.6	4.2	**4.7**	16	9	4	3	4	**3**
LASFile_4	48,471,887	0	f	1.9	3.9	4.9	4.5	**5.0**	46	24	12	9	10	**9**
LDR030828_212242_0	59,672,207	0	f	1.9	3.8	4.8	4.4	**4.9**	57	30	15	12	13	**12**
LDR030828_213023_0	58,414,787	0	f	1.9	3.8	4.8	4.5	**5.0**	56	29	15	12	12	**11**
LDR030828_213450_0	53,215,067	0	f	1.9	3.8	4.8	4.3	**4.9**	51	27	13	11	12	**10**
Lincoln	185,565,975	0	x	1.7	6.1	6.1	6.1	**6.5**	177	106	29	29	29	**27**
line_27007_dd	107,603,879	0	x	2.4	3.5	3.6	**4.0**	3.8	103	42	30	28	**26**	27
MARS_Sample_Filtered_LiDAR	163,225,753	0	x f t	2.8	6.5	7.1	6.3	**8.3**	156	56	24	22	25	**19**
Mount St Helens Nov20 2004	115,737,877	0	x	2.2	12.4	**13.4**	12.9	12.6	110	51	9	**8**	9	9
Mount St Helens Oct4 2004	134,868,035	0	x	1.7	6.1	**6.7**	6.4	6.6	129	78	21	**19**	20	19
ncwc000008	63,161,789	0	f t	1.8	3.1	**3.5**	3.1	3.4	60	34	19	**17**	19	18
Palm Beach Pre Hurricane	51,612,715	0	x	1.6	6.6	**7.0**	6.5	6.8	49	30	7	**7**	8	7
Dallas	104,639,368	1	f t	1.8	5.2	6.2	2.8	**7.5**	100	55	19	16	36	**13**
IowaDNR-CloudPeakSoft-1.0-UTM15N	163,727,279	1	f t	2.5	8.8	8.3	3.3	**11.2**	156	62	18	19	47	**14**
LDR091111_181233_1	54,609,113	1	f	1.8	4.4	5.1	2.8	**5.6**	52	29	12	10	19	**9**
LDR091111_182803_1	54,255,417	1	f	1.8	4.5	4.9	2.8	**5.7**	52	28	12	11	19	**9**
merrick_vertical_1.2	54,609,113	1	f	1.8	4.4	5.1	2.8	**5.6**	52	29	12	10	19	**9**
S1C1_strip021	78,220,943	1	f	2.0	6.2	7.8	3.4	**8.4**	75	37	12	10	22	**9**
Serpent Mound Model LAS Data	91,423,839	1	f	2.2	5.6	6.9	3.3	**8.8**	87	40	15	13	27	**10**
Tetons	104,800,536	1	f t	2.0	4.8	5.0	2.8	**6.0**	100	50	21	20	35	**17**
USACE_Merrick_lots_of_VLRs	101,081,369	1	f t	1.8	4.6	5.2	2.9	**5.5**	96	54	21	19	33	**18**
LAS12_Sample_withRGB_Quick_Terrain_Modeler	99,156,855	2	x	2.7	6.5	error	6.5	**7.4**	95	35	14	error	15	**13**
xyzrgb_manuscript	56,046,269	2	t	2.9	8.5	10.2	9.7	**10.9**	53	18	6	5	5	**5**
autzen-colorized-1.2-3	362,213,959	3	f t	2.1	2.3	5.2	3.1	**6.6**	345	164	148	66	119	**52**
total	2,599,260,453			2.0	4.5	5.9	4.1	**6.4**	2,479	1,210	551	425	610	**388**

TABLE VI
COMPRESSION PERFORMANCE BETWEEN WINZIP (ZIP), WINRAR (RAR), LASCOMPRESSION (LCMP), THE LIZARDTECH LIDAR COMPRESSOR (SID), AND LASZIP (LAZ) FOR TYPICAL, EXPERIMENTAL, AS WELL AS UNUSUAL LAS FILES THAT ARE AVAILABLE AT HTTP://LIBLAS.ORG/SAMPLES.

LAS/LAZ files with any LAStools tool or any application that uses the LASlib API [2] to read or write LAS or LAZ files.

ACKNOWLEDGMENT

The author would like to thank those who have downloaded “LAStools” and sent useful feature requests or bug reports, Howard Butler for suggesting “chunking”, and Tim Loesch from the DNR Minnesota for the first large-scale LAZ campaign. Financial support for version 2.0 of LASzip was provided by Dave Finnegan from USACE Cold Regions Research and Engineering Laboratory and by Hobu Inc. in conjunction with other open source efforts it operates including libLAS and PDAL. Michael P. Gerlek from Flaxen Geo and Howard Butler from Hobu Inc. assisted with the integration of LASzip versions 1.0 and 2.0 into libLAS.

REFERENCES

[1] ASPRS LAS format, “Specifications for the LASer exchange file format,” accessed on November 21th 2011. [Online]. Available: http://www.asprs.org/LAS_Specification
[2] LAStools, “Efficient tools for LiDAR processing.” [Online]. Available: http://www.lastools.org/
[3] libLAS, “A LAS 1.0/1.1/1.2 ASPRS LiDAR data translation toolset.” [Online]. Available: http://www.liblas.org/
[4] OpenTopography, “A portal to high-resolution topography data and tools.” [Online]. Available: http://www.opentopography.org/
[5] Minnesota Department of Natural Resources, “Minnesota state LiDAR collect.” [Online]. Available: ftp://lidar.dnr.state.mn.us/
[6] M. Isenburg, P. Lindstrom, and J. Snoeyink, “Lossless compression of predicted floating-point geometry,” *Computer-Aided Design*, vol. 37, no. 8, pp. 869–877, 2005.
[7] M. Deering, “Geometry compression,” in *SIGGRAPH 95 Conference Proceeisngs*, 1995, pp. 13–20.
[8] C. Touma and C. Gotsman, “Triangle mesh compression,” in *Graphics Interface’98 Proceedings*, 1998, pp. 26–34.
[9] G. Taubin and J. Rossignac, “Geometric compression through topological surgery,” *ACM Transactions on Graphics*, 17 (2), pp. 84–115, 1998.
[10] O. Devillers and P.-M. Gandoin, “Geometric compression for interactive transmission,” in *Proc. of IEEE Visualization 2000*, 2000, pp. 319–326.
[11] M. Isenburg and S. Gumhold, “Out-of-core compression for gigantic polygon meshes,” in *SIGGRAPH 2003 Proceedings*, 2003, pp. 935–942.
[12] J. Peng and C. C. Kuo, “Geometry-guided progressive lossless 3d mesh coding with octree (OT) decomposition,” in *SIGGRAPH ’05 Proceedings*, 2005, pp. 609–616.
[13] M. Isenburg, P. Lindstrom, and J. Snoeyink, “Streaming compression of triangle meshes,” in *Proceedings of the 3rd Symposium on Geometry Processing*, 2005, pp. 111–118.
[14] O. Devillers and P.-M. Gandoin, “Progressive and lossless compression of arbitrary simplicial complexes,” in *SIGGRAPH’2002*, pp. 372–379.
[15] M. Botsch, A. Wiratanaya, and L. Kobbelt, “Efficient high quality rendering of point sampled geometry,” in *Eurographics Rendering Workshop*, 2002, pp. 53–64.
[16] R. Schnabel and R. Klein, “Octree-based point-cloud compression,” in *Eurographics Symposium on Point-Based Graphics*, 2006, pp. 111–120.
[17] M. Waschbüsch, M. Gross, F. Eberhard, E. Lamboray, and S. Würmlin, “Progressive compression of point-sampled models,” in *Eurographics Symposium on Point-Based Graphics*, 2004, pp. 95–102.
[18] S. Gumhold, Z. Karni, M. Isenburg, and H. P. Seidel, “Predictive point-cloud compression,” in *SIGGRAPH ’05 Sketches*, 2005, p. 137.
[19] B. Merry, P. Marais, and J. Gain, “Compression of dense and regular point clouds,” *Computer Graphics Forum*, 25 (4), pp. 709–716, 2006.
[20] LizardTech®, “LiDAR compressor™,” accessed on October 17th 2011. [Online]. Available: http://www.lizardtech.com/
[21] LASCompression, “A lossless compression algorithm for LiDAR datasets,” accessed on November 19th 2011. [Online]. Available: http://gemma.uni-mb.si/lascompression/
[22] D. Mongus and B. Zalik, “Efficient method for lossless LiDAR data compression,” *International Journal of Remote Sensing*, vol. 32, pp. 2507–2518, May 2011.
[23] A. Moffat, R. M. Neal, and I. H. Witten, “Arithmetic coding revisited,” *ACM Transactions on Information Systems*, 16 (3), pp. 256–294, 1998.
[24] A. Said, “Fastac: Arithmetic coding implementation,” accessed on October 30th 2009. [Online]. Available: http://www.cipr.rpi.edu/ said/FastAC.html