# GPU-BASED LOSSLESS VOLUME DATA COMPRESSION

*S. Guthe and M. Goesele*

TU Darmstadt, Germany

## ABSTRACT

In rendering, textures are usually consuming more graphics memory than the geometry. This is especially true when rendering regular sampled volume data as the geometry is a single box. In addition, volume rendering suffers from the curse of dimensionality. Every time the resolution doubles, the number of projected pixels is multiplied by four but the amount of data is multiplied by eight. Data compression is thus mandatory even with the increasing amount of memory available on today's GPUs. Existing compression schemes are either lossy or do not allow on-the-fly random access to the volume data while rendering. Both of these properties are, however, important for high quality direct volume rendering. In this paper, we propose a lossless compression and caching strategy that allows random access and decompression on the GPU using a compressed volume object.

***Index Terms —*** GPU, volume compression

## 1. INTRODUCTION

Just like regular images, most volume data sets are uniformly sampled in three dimensions. However, while the number of pixels in an image increases with a power of two as resolution increases linearly, the number of samples, so called voxels, in a volume data set increases with a power of three. Therefore, a volume data set of $1m^3$ sampled at a resolution of $1mm^3$ per voxel, already contains 1 giga-voxel. Even with today's increase in GPU memory, larger volume data sets fill most of the video memory, leaving little room for additional data to be kept around. However, most data sets contain large virtually homogeneous regions or redundancy, making even lossless compression of the data quite efficient. Such lossless compression has the advantage that it avoids introducing any compression or blocking artifacts. Most existing compression schemes are however, fixed bit rate and therefore lossy or do not allow for random access into the compressed representation. We propose a combined compression and caching scheme for volume data that is able to perform lossless compression on volume data sets while at the same time allowing for efficient random access. We observe typical lossless compression rates on real benchmark data sets of 2.3x to 17x.

## 2. RELATED WORK

Recent surveys such as the Rodríguez et al. [1] show that there is no compression domain volume rendering algorithm that supports at the same time both lossless compression and high quality shading.

**Volume Compression:** While initial compression algorithms for direct volume rendering [2, 3, 4, 5] mostly favor wavelet-based transforms in combination with compression domain rendering, more recent approaches use transformations such as the near-lossless Karhune-Loeve Transform, which can alo be used for decompression during rendering [6]. Another group of compression algorithms that allow for decompression during rendering are the texture compression formats as supported by the hardware (e.g. ASTC [7]). However, since all hardware supported compression schemes are fixed bit rate, they can never be used for lossless compression purposes.

**Coding:** In order to efficiently decompress data on the GPU, we need a compression scheme that easily supports decompression in parallel and compresses small values effectively. Fortunately, the Recursive Bottom Up Complete (RBUC) [8] fulfills both of these needs. RBUC is a hierarchical extension of Elias gamma codes [9] where more than one positive integer is encoded at once.

Other efficient compression options include dynamic Huffman coding [10] and arithmetic coding [11]. Both of these require, however, sequential coding of large data chunks.

## 3. COMPRESSED DATA REPRESENTATION

Unfortunately, efficient lossless compression and random access to individual samples are competing goals so we have to find a good trade-off between these two. Since both extremes—random access to individual voxels and decompression of all data up front—are very impractical, we need to organize the volume in small blocks (so called bricks) that are compressed, decompressed and cached individually. Therefore, our compression approach consists of the following steps: First, we split the volume into small bricks that will be encoded individually. In order to re-create the original data, we need a down-sized volume that contains references to the compressed bricks. For efficiency reasons, multiple references are allowed to point to the same brick. Second, each brick is transformed using a predictor function that exploits local coherence inside the brick as much as possible to produce difference values close to zero. Third, the difference values are encoded using a lossless compression scheme.

### 3.1. Bricking

Splitting volume data into bricks for rendering, so called bricking, is a very common approach. In contrast to hybrid approaches such as the one proposed by Guthe and Strasser [12], we do not have to decompress and upload the data required for rendering up front. Since we store the actual compressed data on the GPU, we will therefore not be able to use the built-in texture interpolation functions, so that there is no need for our bricks to overlap.

Regarding the size of individual bricks, there are several considerations we need to take into account. We have to base the brick size on the fact that compression efficiency goes up with increasing the brick size but random access becomes more expensive. Since there are 32 cooperating threads within a warp, using $4 \times 4 \times 4 = 64$ voxel per brick turned out to be the most efficient choice.

### 3.2. Offset Volume

In order to randomly access individual voxels, we need an index volume that stores the starting address of each compressed brick or an index into an array containing the compressed data at the granularity of bytes. The most basic approach for storing these offsets is to simply use a 3D texture that can be indexed using the raw sample positions. However, due to the limited number of pixel formats, this approach does not perform optimal. Instead, we store the offset into the compacted data structure as a bit field of minimal size for maximum efficiency, e.g. 10 bit per brick if the maximum offset is 1023.

### 3.3. Transforms

Similar to the PNG compression scheme [13], the raw data $raw\,(v)$ of each voxel $v$ within a brick is transformed prior to the actual compression. First, we store the range of a brick as two uncompressed values $min$ and $max$. If $min == max$, we do not need to store any additional data since the entire block is constant. Otherwise, there are currently three different transforms implemented that will be selected on a per brick basis. Note that globally selecting a certain transform reduces the overall compression ratio, especially when selecting subtract minimum/maximum. Also, the per brick selection increases the performance compared to globally picking either gradient prediction or the wavelet transform.

**Subtract Minimum/Maximum:** For subtract minimum (maximum), the predictor for each voxel $v$ is simply the minimum (maximum) of the entire brick and we only need to store the difference. Since the predictor is constant for the entire brick, all voxel can be processed independently. Also, the range of the transformed values is always in the range $[0\,..\,max - min]$.

**Gradient Predictor:** For the first voxel of each brick, we use $\lfloor \frac{min+max}{2} \rfloor$ as a predictor. All other voxel are encoded using a 3D extension of the approach of Paeth [14]. However, the initial predictor is clamped to the valid range of the brick $[min\,..\,max]$, instead of using the neighbor closest to the predicted value.

In order to produce transformed values in the same range as above, we reorder the signed difference values starting with 0, sorted by the absolute value and alternating between negative and positive, e.g. for $[-5\,..\,2]$: $0, -1, 1, -2, 2, -3, -4, -5$.

**Haar Wavelet:** Regarding complexity, the integer Haar Transform [15] is right in between the subtract minimum (maximum) and the gradient predictor schemes. After applying the 3D transform twice, we are left with a single average value of the entire brick, which we store in the same way as the first voxel above and 63 (signed) wavelet coefficients. The wavelet coefficients are stored as unsigned values with the least significant bit denoting the sign, e.g. $0, -1, 1, ...$

### 3.4. Coding

Once the raw data is transformed to maximize the compression efficiency, we can encode the resulting data. While most compression algorithms are inherently serial, the RBUC*n* [8] stores values in a tree-like structure where all children of a node can be decompressed in parallel. Since we have to encode 64 values in each brick, we use a variant of RBUC8 where the number of children $s_i$ at level $i$ of the tree is fixed to 8 rather than increasing as we move closer to the root of the tree. The actual construction of the codeword for a single brick is as follows. First, we order all transformed values according to their Morton code

$$\begin{aligned} c_0 \;=\; & [(0,0,0)\,,(1,0,0)\,,(0,1,0)\,,(1,1,0)\,,\;\ldots\;, \\ & (3,2,3)\,,(2,3,3)\,,(3,3,3)] \end{aligned}$$

Next we compute the maximum number of bits required to represent each value over each group of 8 values:

$$c_1\,(x) = \lceil \log_2\left(\max\left(c_0\,(8x)\,,\;\ldots\;,c_0\,(8x+7)\right)\right)\rceil$$

Each $c_1$ now corresponds to one of the sub-bricks. Finally, we calculate the number of bits $c_2$ required for representing all $c_1$ values similar to the equation above. The actual codeword is then constructed as follows:

- One byte containing $c_2$.
- One set of $c_2$ bytes containing the values $[c_1\,(0)\,,\;\ldots\;,c_1\,(7)]$ each using $c_2$ bits.
- Eight groups of $c_1\,(i)$ bytes containing the values $[c_0\,(8i)\,,\;\ldots\;,c_0\,(8i+7)]$ each using $c_1\,(i)$ bits.

Note that each of the items in the list above, including each of the eight groups, is aligned to byte boundaries making decompression as efficient as possible.

## 4. DECOMPRESSION AND CACHING

In order to allow fast access, we need to be able to decompress data efficiently. We furthermore cache decompressed values on a per brick basis to maximize performance.

### 4.1. Parallel Decompression

The warp based parallel decompression first needs to expand a compressed brick into the corresponding encoded values. If the entire brick is constant, i.e. $min = max$, the actual decompression is skipped. Otherwise, we use the following approach:

- All threads read $c_2$.
- The first 8 threads $i \in [0..7]$ of a warp read $c_1\,(i)$ and propagate the values to other threads in the warp accordingly.
- Each thread $i \in [0..31]$ reads $c_0\,(i)$ and $c_0\,(i+32)$.

Note that all communication between threads uses the shuffle intrinsic that transfers data within a warp without using shared memory.

Unless $min == max$, we still need to do the inverse of the transform used during encoding. As the decoded values might not be in the correct thread for the inverse transform, we again need to use up to two shuffle instructions.

For the *Subtract Minimum/Maximum* case the inverse transform can easily be done in parallel as the predictor is always constant, i.e. $min$ or $max$.

The *Gradient Predictor* is a little more complicated since the prediction for each voxel depends on the original raw values. Thus the inverse transform propagates like a wave front through the brick and requires a total of 10 steps where each thread is active twice. More specific, in the first step, only $raw\,(0,0,0)$ can be recovered by thread 0. In the second step, threads 1, 4 and 16 calculate $raw\,(0,0,1)$, $raw\,(0,1,0)$ and $raw\,(1,0,0)$ respectively. In general the value $raw\,(i_0,i_1,i_2)$ is recovered in step $i = i_0 + i_1 + i_2 + 1$ by thread $j = (i_0 + 4i_1 + 16i_2)\,\%32$.

In case of the *Haar Wavelet*, the first inverse transform uses the first 4 threads of the warp only due to it's hierarchical nature. The second inverse transform uses all 32 threads. Once the entire brick is decompressed, each thread can either access the data using the shuffle intrinsic or the result can be cached in shared memory.

### 4.2. Per Warp Caching of Voxel Bricks

We decided to only share cached data inside a warp rather than across a whole block for the following three reasons: First, for larger volume data sets, we can assume that the footprint of a voxel projected onto the screen roughly corresponds to a pixel. Thus, the

| data set | dimensions | bpv | size in MB |
|---|---|---|---|
| Carp | $256 \times 256 \times 512$ | 16 | 64.00 |
| Bunny | $512 \times 512 \times 361$ | 16 | 180.50 |
| Porsche | $559 \times 1023 \times 347$ | 8 | 189.24 |
| C. Present | $492 \times 492 \times 442$ | 16 | 204.07 |
| C. Tree | $512 \times 499 \times 512$ | 16 | 249.50 |
| Stag Beetle | $832 \times 832 \times 494$ | 16 | 652.23 |
| Vis. Human | $1600 \times 1600 \times 1882$ | 32 | 18,378.91 |

Table 1: Data set dimensions and raw data sizes.

coherence between rays of different warps is very low. Second, for small data sets, the amount of decompression that needs to be done per warp is very small to start with. Third, if threads only communicate within a warp, there is no need to synchronize across the entire block. Not having this synchronization overhead leads to an increased in performance.

**Cache Layout:** The cache is fully associative, stored in shared memory and consists of two parts, the tags (the value stored in the initial offset volume) and the actual decompressed data. Depending on the size of the indices in the offset volume, the tags can be stored as 32 or 64 bit each. Storing less than 32 bit does not increase the performance as these will either be stored in registers or shared memory. The decompressed data depends on the number of bit per component and is either 64, 128 or 256 byte per brick. With up to 32 cache entries per warp, we end up with up to $8.25kB$ of shared memory per warp.

**Replacement Scheme:** Ideally, we would like to cache all bricks that are used for a single sample in the entire warp. Since this may exceed the number of cache entries available, we use the following scheme that efficiently implements an LRU replacement strategy:

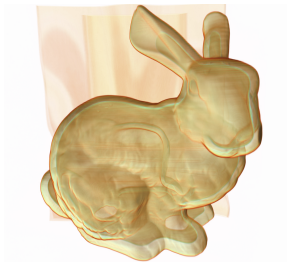
- Mark all cached bricks that are being used for the current sample as *active* and read all necessary data from those bricks.
- For each required brick that is not in the cache, replace the first *inactive* entry and mark it as *active* and *new*. Also read all the data currently required.
- If there are no more inactive entries, mark all *active* entries as *inactive* and all *new* entries as *active* only.

This causes newly decompressed bricks to stay in cache as they are more likely to be used again than bricks that have been decompressed for prior samples. The total amount of information per cache entry required for this replacement scheme is just 2 bit and the first available entry can be found by simply using the count-leading-zeros (clz) intrinsic.

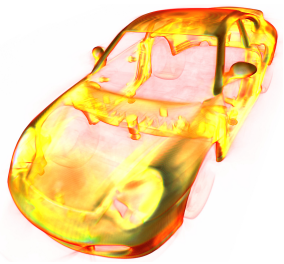
## 5. RESULTS

In order to evaluate the whole algorithm, we analyze all parts of the compression and decompression pipeline using real-world data and actual volume rendering algorithms. All tests use an NVIDIA GeForce GTX TITAN X with CUDA 7.5, diver version 355.98, and an extension of the volume rendering example found in the CUDA SDK.

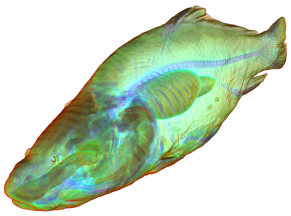
We tested a variety of data sets, including 8 bit per voxel (bpv) and 16 bpv scalar data as well as 32 bpv ARGB data (separately compressed component), ranging from a couple of MB up to 18 GB of raw data, see Figure 1 and Table 1. Note that the volume data sets do not necessarily contain cubic voxel but usually have an actual voxel extend as defined by the data set, e.g. $0.33\,mm \times 0.33\,mm \times 1.0\,mm$ for the Visible Human 3.



(a) Bunny [16]

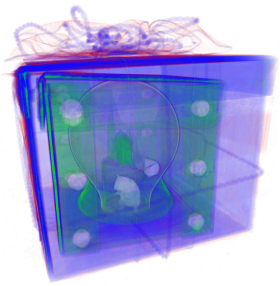


(b) Porsche [16]

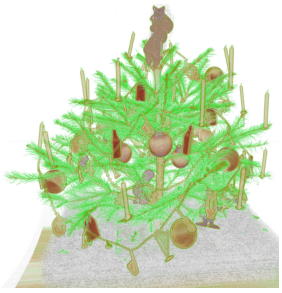


(c) Carp [16]

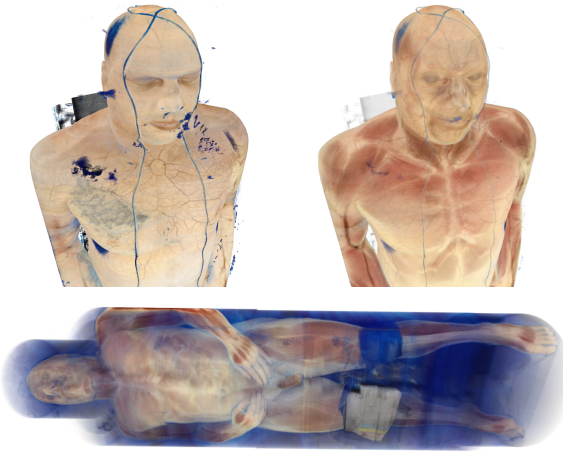
(d) Stag Beetle [17]



(e) Christmas Present [18]



(f) Christmas Tree [19]



(g) Visible Human Male (cryosection registered against CT) [20]

Figure 1: Data sets used for evaluation.

**Compression:** Even though we need to store an additional index volume (see Table 2), the overall compression rate is very close to the entropy of the encoded values. However, since our compression scheme assumes a normal distribution over all values with a maximum at zero after the transform step, it suffers from any kind of skewed distribution. In this case the compressed voxel data is slightly larger than its entropy. Thus our compression is not only very well suited for parallel decompression but also outperforms an arithmetic coding of the same data.

**Caching:** The cache hit rate is very important since decom-

| data set | compressed bit per voxel | | |
|---|---|---|---|
| | our | | entropy |
| Carp | 5.48 | (5.08) | 6.88 |
| Bunny | 5.66 | (5.25) | 6.55 |
| Porsche$^{8\,\mathrm{bpv}}$ | 1.48 | (1.09) | 2.79 |
| Christmas Present | 6.02 | (5.60) | 5.33 |
| noise removed | 1.61 * | (1.25) | 1.46 |
| Christmas Tree | 4.77 | (4.34) | 4.95 |
| noise removed | 4.11 | (3.70) | 3.24 |
| Stag Beetle | 0.70 * | (0.40) | 0.66 |
| Visible Human | 13.15 | (12.64) | 22.86 |
| clipped $^{32\,\mathrm{bpv}}$ | 11.08 | (10.56) | 20.17 |

Table 2: Compressed size compared against the entropy in bit per voxel (with and without offset volume). *Offset volume compacted using second level of indirection.

| data set | $1920 \times 1080$ | | |
|---|---|---|---|
| | hit | dec. | speedup |
| Carp | 99.61% | 6.68 | 190.60 |
| Bunny | 99.65% | 7.41 | 142.49 |
| Porsche | 99.54% | 4.01 | 107.74 |
| Christmas Present | 99.92% | 1.37 | 654.70 |
| Christmas Tree | 99.61% | 6.68 | 126.67 |
| Stag Beetle | 99.97% | 0.22 | 1668.45 |
| Visible Human | 98.93% | 0.57 | 45.57 |

Table 3: Average cache efficiency during rendering using tri-linear interpolation. Decompression measured per frame relative to the size of the data set. Cache size is 23 entries except for Porsche (32) and Visible Human (11).

pression is a lot slower than just reading values from shared memory. Without the cache, we would spend more than $99.9\%$ of the time just decompressing data. As seen in Table 3, the hit rate for our cache is always above $98\%$ at HD resolution. This corresponds to a performance increase of at least 40 times compared to rendering without the cache.

## 6. CONCLUSION & FUTURE WORK

In this paper, we presented a fast and parallel lossless compression scheme that can be used in direct volume rendering. It consists of a warp centric decompression and caching scheme. Since all functionality can be hidden behind a simple function call, it can easily be deployed into existing rendering algorithms. Our approach allows us to render volume data sets that would not fit onto a GPU in uncompressed form while avoiding any artifacts due to lossy compression. In the future, we would like to explore possible extensions towards multi-resolution volume rendering for output sensitive performance.

## 7. REFERENCES


[1] M. Rodríguez, E. Gobbetti, J. Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter, "A Survey of Compressed GPU-Based Direct Volume Rendering," in *Proc. Eurographics*, 2013.

[2] T. Kim and Y. Shin, "An Efficient Wavelet-Based Compression Method for Volume Rendering," in *Proc. Pacific Graphics*, 1999.

[3] I. Ihm and S. Park, "Wavelet-Based 3D Compression Scheme for Very Large Volume Data," in *Graphics Interface*, 1998.

[4] K. Nguyen and D. Saupe, "Rapid High Quality Compression of Volume Data for Visualization," in *Computer Graphics Forum*, 2001, vol. 20.

[5] F. Rodler, "Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data," in *Proc. Pacific Graphics*, 1999.

[6] N. Fout and Kwan-Liu M., "Transform Coding for Hardware-accelerated Volume Rendering," *Trans. on Visualization and Computer Graphics*, vol. 13, no. 6, Nov 2007.

[7] J. Nystad, A. Lassen, A. Pomianowski, S Ellis, and T Olson, "Adaptive Scalable Texture Compression," in *Proc. ACM SIGGRAPH/Eurographics Symposium on High Performance Graphics*, 2012.

[8] A. Moffat and V. Anh, "Binary Codes for Non-Uniform Sources," in *Proc. Data Compression Conference*, 2005.

[9] P. Elias, "Universal Codeword Sets and Representations of the Integers," *IEEE Trans. on Information Theory*, vol. 21, no. 2, Mar 1975.

[10] D. Knuth, "Dynamic Huffman Coding," *Journal of Algorithms*, vol. 6, no. 2, 1985.

[11] I. Witten, R. Neal, and J. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, 1987.

[12] S. Guthe and W. Strasser, "Advanced Techniques for High-Quality Multi-Resolution Volume Rendering," *Computers & Graphics*, vol. 28, no. 1, 2004.

[13] D. Duce, "Portable Network Graphics (PNG) Specification (Second Edition)," World Wide Web Consortium, Recommendation REC-PNG-20031110, 2003.

[14] A. Paeth, "Image File Compression Made Easy," in *Graphics Gems II*, 1994.

[15] A. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Wavelet Transforms that Map Integers to Integers," *Applied and Computational Harmonic Analysis*, vol. 5, no. 3, 1998.

[16] S. Röttger, "The Volume Library," http://lgdv.cs.fau.de/External/vollib/.

[17] E. Gröller, G. Glaeser, and J. Kastner, "Stag Beetle," http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/.

[18] C. Heinzl, "Christmas Present," http://www.cg.tuwien.ac.at/research/publications/2006/dataset-present/.

[19] A. Kanitsar, T. Theußl, L. Mroz, M. Srámek, A. Bartrolí, B. Csébfalvi, J. Hladuvka, D. Fleischmann, M. Knapp, R. Wegenkittl, P. Felkel, S. Röttger, S. Guthe, W. Purgathofer, and E. Gröller, "Christmas Tree Case Study: Computed Tomography As a Tool for Mastering Complex Real World Objects with Applications in Computer Graphics," in *Proc. IEEE Visualization*, 2002.

[20] M. Ackerman, "The National Library of Medicine: The Visible Human Project," http://www.nlm.nih.gov/research/visible/visible_human.html.