Fast Compressed Segmentation Volumes for Scientific Visualization

Max Piochowiak, and Carsten Dachsbacher

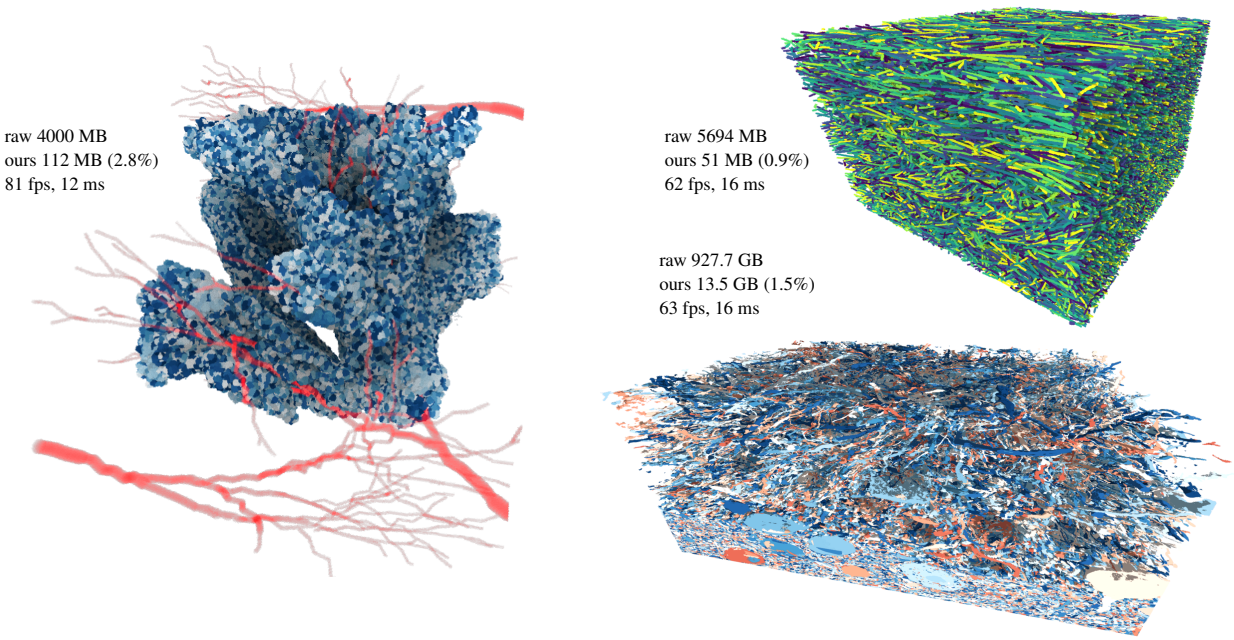

Fig. 1: Our lossless compressed segmentation volumes encode bricks in a multi-resolution fashion followed by entropy coding. Our technique achieves strong compression ratios and provides fast decompression and adaptive level-of-detail. The images show interactive renderings of data sets up to 927 GB uncompressed size (*raw*) at 1920×1080 resolution computed using ray marching with shadows. The raymarching step size is set to match the size of a voxel. Transfer functions (mapping labels to color and opacity) and clipping can be changed interactively.

Abstract— Voxel-based segmentation volumes often store a large number of labels and voxels, and the resulting amount of data can make storage, transfer, and interactive visualization difficult. We present a lossless compression technique which addresses these challenges. It processes individual small bricks of a segmentation volume and compactly encodes the labelled regions and their boundaries by an iterative refinement scheme. The result for each brick is a list of labels, and a sequence of operations to reconstruct the brick which is further compressed using rANS-entropy coding. As the relative frequencies of operations are very similar across bricks, the entropy coding can use global frequency tables for an entire data set which enables efficient and effective parallel (de)compression. Our technique achieves high throughput (up to gigabytes per second both for compression and decompression) and strong compression ratios of about 1% to 3% of the original data set size while being applicable to GPU-based rendering. We evaluate our method for various data sets from different fields and demonstrate GPU-based volume visualization with on-the-fly decompression, level-of-detail rendering (with optional on-demand streaming of detail coefficients to the GPU), and a caching strategy for decompressed bricks for further performance improvement.

Index Terms—Segmentation volumes, lossless compression, volume rendering.

1 Introduction

In many applications such as in materials science [49], connectomics [12], or computational biology [7], voxel-based segmentation volumes represent how individual regions of interest occupy space in the observed volume by assigning a label to each voxel. These volumes can, for example, be obtained from segmenting scalar or multivariate volume data in a preprocessing step [38], but they can also be the primary output from simulations [7]. Segmentation volumes are of fundamental importance when complex structures in large volumes are studied by visual exploration [1]. However, as with large volume data in general the storage requirements can be challenging, e.g. when time-series or large ensembles of simulations are computed and stored, or when interactive visualization requires a large portion of a data set to reside in GPU memory for efficient rendering. Compression techniques can alleviate this problem. In fact a large variety of techniques for volumes storing quantitative data exists [5], but such techniques are not directly applicable or beneficial to segmentation volumes as they are meant to represent scalar/vectorial signals. In practice, segmentation

• *Max Piochowiak and Carsten Dachsbacher are with Karlsruhe Institute of Technology. E-mail: {max.piochowiak | dachsbacher}@kit.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

volumes are often sliced followed by general-purpose image compression techniques. However, this approach is not suited for on-the-fly decompression during visualization and typically less effective than tailored techniques. These, in contrast, exploit the characteristics of segmentation volumes. Compresso [36], for example, encodes the boundaries of segmented regions followed by the Lempel–Ziv–Markov chain compression algorithm (LZMA). With this combination it achieves high compression ratios, but lacks high decompression speed and the possibility to decompress individual parts of the volume both of which are important for interactive visualization.

In this paper we propose a novel lossless compression technique for segmented volumes. Our method achieves high compression and decompression speed, strong compression ratios, and is lightweight and well-suited for GPU-based decompression on-the-fly. Our compression scheme is based on a per-brick multi-resolution representation of the segmentation volume which exploits the presence of homogeneous label neighborhoods and effectively encodes the boundaries in between. The output of this step is a list of labels and a sequence of operations with which the brick can be reconstructed. For further compression, we store the sequence of operations using a fast asymmetric numeral systems entropy coding scheme (rANS [17]). In summary, our contributions are:

- a lossless compression technique for segmented volumes with strong compression ratios, little memory overhead, and fast execution times,
- a multi-resolution representation for level-of-detail and a sufficiently fine granularity for accessing compressed data,
- a parallel, GPU-friendly decompression and caching strategy for interactive visualizations of compressed segmentation volumes, also supporting on-demand streaming of detail information.

We will first overview related work on compression for volume rendering. In Sec. 3 we explain our mulitresolution encoding and decoding scheme for segmented volumes and the subsequent entropy coding. Thereafter we detail the decompression in the context of interactive visualization and GPU-rendering (Sec. 4). We evaluate our method on multiple data sets and report compression rates and rendering performance (Sec. 5).

2 Related Work

Volume and image compression have been an active area of research for decades [24, 42]. In this overview, we focus on works in the scope of compression of voxel data and segmentation volumes for rendering and scientific visualization.

Compression of Quantitative Volume Data Rodríguez et al. [5] and Beyer et al. [10] provide comprehensive overviews of compression for scalar volumes. As the input data is often noisy, many existing techniques for scalar volumes perform lossy compression, e.g. using wavelets [25] or neural compression [32, 47]. As a GPU implementation of the OpenVDB data structure, NanoVDB supports quantization based compression [40]. Lossless techniques exist as well, ranging from run-length encoding [4, 43] [34], wavelet transforms [21], or Huffman [19] and other entropy coders [31]. In principle, some of these techniques can be applied for segmentation volumes, but since they are tailored to quantitative data they perform suboptimally. Still, individual building blocks can be shared, for example, efficient GPU-based implementations for entropy and range coders exist [48]; we also make use of rANS-coding [17] in our method. Many volume compression techniques make use of hierarchical representations: Sparse voxel octrees [29] and its extension to sparse voxel directed acyclic graphs (SVDAGs) [27] are widely used for efficient lossless volume compression in rendering [13, 33, 46]. SVDAGs reuse sub-trees of an initially constructed octree; extensions of the original scheme for binary data can be use to compress arbitrary attributes, making it suitable for a wider range of applications [16]. Dado et al. [15] use compressed palettes of voxel attributes that are accessed with an indexing scheme over the graph edges. Mados et al. [35] allow replacing homogeneous subregions in DAGs with arbitrary constant values and use variable bit length encoding on voxel attributes stored in leaf nodes. Note that these methods are optimized for scalar volume data, and they rely on sparsity in the input data to achieve compact representation, a property that segmentation volumes are typically lacking.

Segmentation Volume Compression Segmentation volumes represent a piece-wise constant, integer-valued function and thus have very different characteristics than the aforementioned volumes. Consequently, specialized compression techniques have been developed. Neuroglancer [20] splits segmentation volumes into bricks, and each brick is represented by a palette of the contained labels plus one index into the palette for every voxel. While the approach is fast and well-suited for direct rendering, it does not achieve competitive compression rates. Compresso [36] also uses a brick-wise encoding and determines a set of (binary) templates to represent region boundaries which is then used for encoding. The final strong compression rate is only reached by using a global LZMA compression; however, this makes it unsuitable for GPU-based rendering and decoding of individual bricks. Our method also compresses individual bricks, however, we avoid a compression of the entire data stream and can inherently decode bricks up to a desired level-of-detail. The Mixture Graph [3] is a representation of segmentation volumes designed for efficient rendering. It offers a multi-resolution tree hierarchy containing packed label histograms for precise color filtering. They use graph compression of histogram factorizations to reduce the memory for their representation. While this technique offers a multi-resolution representation, our technique results in an order of magnitude smaller compressed size and faster (de)compression.

Large-Scale Segmentation Volume Visualization Direct rendering of large volume data usually relies on out-of-core methods and streaming [9, 11]. Caching and memory virtualization are used to hide the latency of CPU-to-GPU streaming and to allow direct access of relevant data for the renderer [23]. In our exemplary implementation, we also use caching of decompressed volume bricks. This cache, in particular its implementation on GPUs, is inspired by the Shading Atlas Streaming [39] which proposes a texture cache for rendering on untethered virtual reality devices. Rendering of segmentation volumes is often carried out alongside general volume rendering pipelines [8, 14] or within volume segmentation pipelines [2, 6] as well as in systems used for the analysis and information visualization of label region attributes [45, 49]. Agus et al. [1] use dimensionality reduction and topological analysis for guided transfer function design for segmentation volumes. For large volume rendering, Beyer et al. [11] combine probabilistic with exact representations within a hierarchical data structure for fast culling and querying of segmentation data. Surface extraction and rendering can also serve as an alternative to volume-based rendering of segmentation data [28, 30, 41]. In our work, we focus on direct GPU-based rendering of large segmentation volumes. We show that large volumes can be visualized interactively even on modest hardware thanks to strong compression rates and fast decompression, but we also demonstrate CPU-to-GPU streaming of the finest levels of detail for data exceeding the available GPU memory.

3 Compressed Segmentation Volumes

In this section we will describe our compressed segmentation volumes (CSVs). In order to achieve fast and parallel (de)compression with sufficiently fine granularity, our (de)compression operates on volume bricks (Sec. 3.1) which are encoded in a multi-resolution fashion (Sec. 3.2), followed by entropy coding (Sec. 3.3).

3.1 Segmentation Volume Bricks

We assume that the input segmentation volumes store a label (an integer value) per voxel and compress individual bricks of the volume separately. We further assume that these bricks all have the same size of b^3 labels with $b = 2^N$, $N \in \mathbb{N}$, i.e. input data might be padded to multiples of b in each dimension.

Our encoding of each brick begins with building a resolution pyramid of $log_2 b + 1$ levels. We denote the finest level storing b^3 labels as L_0, and successively compute the coarser levels L_l, $0 < l \leq N = log_2 b$.

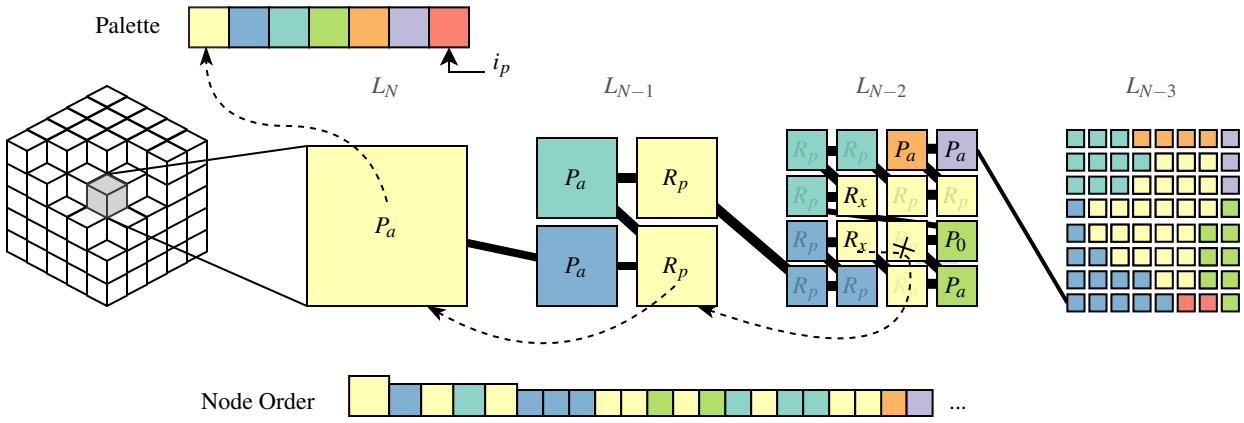

Fig. 2: A 2D example of the multi-resolution encoding of a brick (different labels shown as colors). First the resolution pyramid with levels $L_N, L_{N-1}, \ldots$ is built and its grid nodes (starting from the root node, largest/left) are processed coarse-to-fine and along the Morton Z-curve within a level. The traversal order for this brick is shown on the bottom. The result of the encoding is a palette of labels (possibly with duplicates) and a sequence of operations, one for each node. These operations (P_a, R_p, $R_{\{x,y,z\}}$, P_0, and P_δ) define how labels are assigned to nodes, e.g. by reading a label from the palette or reusing the label of its parent or neighbors.

In each step the resolution is halved along each axis, i.e. L_l stores $(b/2^l)^3$ labels, and each voxel in L_l overlaps 2^3 voxels in L_{l-1}. For the label of a voxel in L_l we assign the most frequent label in the corresponding 8 voxels in L_{l-1}. As this resolution pyramid essentially forms an octree, we will refer to voxels in the pyramid as *nodes* when describing the encoding and decoding (see Fig. 2). We generate the pyramid explicitly for a brick during encoding which requires about 14% additional temporary memory. The encoded representation implicitly contains the multi-resolution representation; decoding does not require more memory than necessary to store the $(b/2^l)^3$ labels of the desired level-of-detail.

3.2 Brick Encoding

The key to a good compression is to compactly encode the assignment of labels to voxels. Our encoding can leverage local homogeneity, e.g. by copying labels from parent or neighbor nodes, and only rarely reading new labels from a separate list which we call the *palette*.

Our encoding, and likewise later the decoding, begins with the coarsest level L_N, which represents the root node of the octree. The label of this node is the first label stored in the palette; we denote the index of the last used palette entry as i_p which is initialized to zero. The respective child nodes in $L_{N-1}, L_{N-2}, \ldots$ then need to be processed in a defined order, for which we use a Morton Z-curve in each level; a concatenation of the Z-curves yields the enumeration of all octree nodes (Fig. 2). The core idea of our encoding is that the label of the next node in this order often can be determined by a simple operation, such as assigning the same label as the parent node, or reading the next label stored in the palette; the result of a brick encoding then becomes a sequence of operations and the associated palette. In the following, we introduce and discuss the individual operations we have chosen. The selection is a result of investigating typical configurations in the resolution pyramids and compression experiments; it comprises the following operations:

- **Parent reuse R_p:** assign the label of the (coarser) parent node to the next node. Note that the processing order of nodes guarantees that the parent node's label is known. For parent nodes we chose the most frequent label among its children, consequently this operation is often applicable.
- **Palette Advance P_a:** increase i_p, read the label at the new index from the palette, and assign it to the next node.
- **Neighbor Reuse R_x, R_y, R_z:** these operations are used to reference nodes adjacent to the next node and assign their label. Note that we reference nodes *outside* the 2^3 block of sibling nodes only (see Fig. 3). This operation can refine boundaries by "pulling in" the label from neighboring regions. Thus the operations $R_{\{x,y,z\}}$ only define along which axis we reference while the direction is implicit. If the referenced node has a later Z-index, however, it is not yet decoded. In this case, we reinterpret a neighbor reuse as a reference to the neighbor's parent whose label is known. Neighbor nodes in other bricks are not referenced. Fig. 2 illustrates such an operation used to define the shape of the yellow region.
- **Palette Back References P_0, P_δ:** When reuse-operations are not applicable, palette advance operations (which require storing another label) can be avoided by back-referencing a previously used palette entry. P_δ references the palette entry at index $i_p - \delta - 1$ ($\delta \in [0, 15]$ is stored as 4-bit value), and the special case operation P_0 indexes the last used palette entry i_p; the respective entry is assigned to the next node.

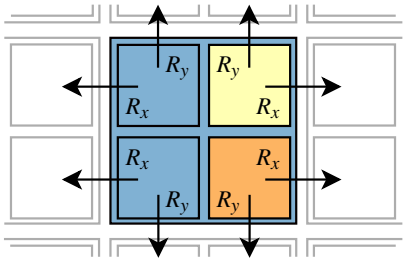

Fig. 3: The operations $R_{\{x,y,z\}}$ assign the same label to the currently processed node as found in an adjacent nodes. They define the axis along which this neighbor is found, the direction always points outside the current 2^3 block of voxels (2^2 in this 2D-example) with the same parent node.

The children and children's children of nodes on finer levels often represent the interior of homogeneous regions and thus carry the same label. To not store redundant operations, we output one additional *stop bit* for every node to indicate whether or not further operations follow for the respective nodes in finer levels.

Discussion In our experiments we considered additional operations, such as reusing diagonal neighbors or grandparents, but found that small operation sets typically yield better results. We also considered resolving references $R_{\{x,y,z\}}$ to not yet decoded neighbors, but this leads to possibly long chains of dependencies. Given that referenced neighbors always have a different parent than the processed node and

Algorithm 1 Decoding a brick up to a target level-of-detail t.

Input brick *operation stream* and *palette*, target LOD t
Output array *out* containing decompressed brick in LOD t

1: $i_p \leftarrow 0$ ▷ palette read index
2: $out[0 \dots 2^{3(N-t)}] \leftarrow \{\varnothing \dots \varnothing\}$ ▷ initialize output array
3: $out[0] = palette[i_p]$
4: **for** $l \in [N \dots (t+1)]$ **do** ▷ coarse to fine
5: **for all** already decoded node (spacing 2^{l-t}) in Z-order **do**
6: $i \leftarrow$ index of the node in *out*
7: **if** $out[lastChildOf(i)] \neq \varnothing$ **then continue** ▷ constant area
8: $parent \leftarrow out[i]$ ▷ store parent for next 8 nodes
9: **for all** child node (spacing 2^{l-t-1}) in Z-order **do**
10: $j \leftarrow$ index of the child node in *out*
11: $(op,\ stop) \leftarrow readNextOperationAndStopBit()$
12: **switch**(op)
13: **case** R_p : $out[j] \leftarrow parent$
14: **case** $R_{\{x,y,z\}}$: $out[j] \leftarrow neighbor\ value$
15: **case** P_a : $out[j] \leftarrow palette[++i_p]$
16: **case** P_0 : $out[j] \leftarrow palette[i_p]$
17: **case** P_δ : $out[j] \leftarrow palette[i_p - \delta]$
18: **if** stop **then**
19: fill j's entire sub-block with $2^{3(l-t-1)}$ labels in *out*

Algorithm 2 Encoding a brick into a palette and sequence of operations.

Input original *brick* voxels from volume
Output brick *operation stream* and *palette*

1: $pyramid \leftarrow$ brick's multi-resolution pyramid to encode
2: $palette \leftarrow$label of pyramid's root node
3: **for** $l \in [N \dots 1]$ **do**
4: **for all** node on level l (spacing 2^l) in Z-order **do**
5: $i \leftarrow$ index of the current node
6: **if** $pyramid[i]$.constantChildren **then continue**
7: $parent \leftarrow pyramid[i]$.label ▷ parent for next 8 nodes
8: **for all** child node (spacing 2^{l-1}) in Z-order **do**
9: $j \leftarrow$ index of current child node
10: $L \leftarrow pyramid[j]$.label
11: $stop \leftarrow pyramid[j]$.constantChildren
12: $op \leftarrow bestOperation(parent, pyramid, palette, L)$
13: **if** $op = P_a$ **then** $palette$.push(L)
14: **if** $op = P_\delta$ **then** output δ
15: output (op, stop)

that those parents often carry the correct label this does not noticeably impact the compression. Lastly, note that palettes can still contain duplicates, e.g. when a label has previously been used in the decoding of a brick, but cannot be referenced with P_δ.

Our encoding operates along the Morton Z-curve, and only P_0 and P_δ refer to previous nodes along the curve. However, the traversal order determines which neighbor references by $R_{\{x,y,z\}}$ have already been en/decoded. As an alternative we tried Hilbert curves which led to almost identical compression rates. While each (non-boundary) node can still reference 3 neighbors on the current level *on average*, their number and the directions to valid references depend on the position along the Hilbert curve (in contrast to Morton Z-curves). The more costly evaluation of the Hilbert curve and the additional cost for referencing result in 2 to 3 times longer compression times. The supplemental material contains a discussion in more detail.

3.3 Entropy Coding of Operation Sequences

The representation of a brick as a sequence of operations already reduces storage, but each operation occupies at least 4 bits (7 different operations plus stop bit; in case of P_δ, 4 additional bits for δ). As the frequencies of operations are highly imbalanced (see Fig. 8) we apply an entropy coding to further reduce storage. We found that using range asymmetric numeral systems (rANS) [17] is a good compromise between Huffman coding (fast, but suboptimal because of the fixed number of bits per symbol) and arithmetic coding (slower). We directly use the sequence of 4-bit nibbles as data stream. Interestingly, the frequencies of these nibbles are extremely similar across the bricks of an entire segmentation volume (see Sec. 5.1 for details). This enables us to quickly determine static, well-suited frequency tables for a data set, and also later efficiently perform the rANS-decompression and execution of operations in one go and in parallel for the volume's bricks. Note that we create two frequency tables per data set: one for the interior nodes (levels $L_n..L_1$), and one for leaf nodes (level L_0) whose stop bits are always 0. Of course parallel (de)compression is also possible with individual frequency tables per brick or adaptive frequencies, but this requires additional storage for the tables or results in worse compression ratios when frequencies need to adapt to the data stream first.

3.4 Encoding and Decoding Implementation

In this section we discuss important algorithmic details and begin with the decoding of a brick from a given palette and sequence of operations and stop bits.

Decoding The decoding begins with the coarsest level L_N and can be performed up to a desired target level-of-detail $L_t, t = N..0$. This eventually results in a block of $2^{3(N-t)}$ labels. Prior to decoding, the memory for this 3D-array output is allocated and the decoding is then performed in-place: While processing a level $L_l, l = N..t$, its labels are stored with a spacing of 2^{t-l} in the output array; all entries are filled when the decoding is complete. Fig. 4 shows a 2D-example with three intermediate decoding steps. Algorithm 1 details the decoding procedure of a single brick. After initialization (lines 1-3) it processes one level-of-detail after another, beginning with the coarsest level L_N (line 4). Line 5 loops over the $2^{3(N-l)}$ nodes on level l in the Morton Z-curve order. If a label has already been assigned to all of its child nodes (line 7), this is because a stop bit has been set on a coarser level (lines 18-19) and no further decoding for this node's children is required[1]. Otherwise, the node's label is temporarily stored in *parent* (line 8), and its eight child nodes are decoded by reading the next 4-bit tuple of operation and stop bit (line 11). The operation determines the label of the next child node (lines 12-17). If the stop bit is set, the entire sub-block of $2^{3(l-t-1)}$ labels is set (lines 18-19). Note that we overwrite the previously decoded coarser LODs until we reach t, but at the expense of higher memory consumption overwriting is not mandatory per se.

Encoding Similar to decoding, all bricks can be encoded in parallel; the pseudo-code is given by Algorithm 2. The encoding of a brick begins with computing the resolution pyramid (*pyramid*) which requires about 14% additional temporary storage for the brick (line 1), similarly to a 3D mipmap [50]. The pyramid contains each node's reference label, and if the node's subtree is constant, the stop bit is set (line 11). If there is an ambiguity when determining the most frequent child label for a pyramid node, we chose the one occurring first within the child node array. The encoding is performed analogous to the decoding,

[1]This test is performed redundantly down to leaf nodes. However, homogeneous regions are typically not large, i.e. stop bits are typically set at finer levels and the overhead remains small. Still the use of stop bits avoids storing superfluous operations for many nodes.

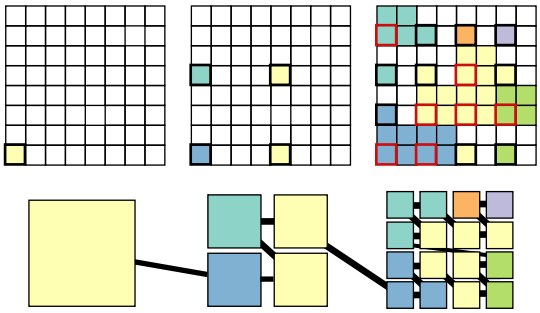

Fig. 4: This example shows three decoding steps $L_N, ..., L_{N-2}$ of a brick (in 2D). The grids (top) show the in-place decoding and are sized to store the brick at target LOD $t = N-3$, i.e. $2^{3(N-t)}$ labels ($64 = 2^{2(N-t)}$ in 2D). The nodes of the resolution pyramid and the Z-curve order is shown on the bottom. Nodes highlighted in red have a stop bit set and assign their label to all their child nodes; for these no further decoding operations are required.

but of course it determines which operation is suitable to yield the label of each child node (line 12). For this, *bestOperation* tests for the first possible operation in a fixed order which we found a good fit to their typical frequencies: $R_p, R_x, R_y, R_z, P_0, P_\delta, P_a$. Sticking to this fixed order also additionally skews the operations' frequencies favorably for entropy coding. For any P_δ we directly output δ as an additional 4-bit entry to the encoding stream (line 14).

Note that when compressing a segmentation volume, we typically feed the output nibbles of a brick directly into the entropy coder. For this, prior to the actual encoding, we perform a quick prepass over a subset of bricks (in our examples every 512^{th} brick, or every 4096^{th} for large data sets) to determine the two frequency tables of nibbles for both interior and leaf nodes. Using two frequency tables has no performance impact on the rANS-encoding, and the additional storage is negligible.

4 Visualization of Compressed Segmentation Volumes

In this section we describe how compressed segmentation volumes (CSVs) can be used with raymarching for volume visualization. We assume that transfer functions (TFs) are used which map labels to color and opacity; note that the design of TFs is orthogonal to our work. Our exemplary raymarcher works similar to other bricked volume visualizations and efficiently supports empty-space skipping of bricks that are invisible due to the TF. Bricks are decompressed with the required level-of-detail (LOD) on demand and then stored in a cache in order to facilitate fast accesses during raymarching and to exploit temporal coherency in the camera movement (and thus in visible bricks and LODs). For an overview of basic raymarching and volume shading techniques we refer the reader to Jönsson et al. [26]. We detail the technical details of our implementation in Sections 4.1 and 4.2 (see also Fig. 5). The next paragraphs provide a high level summary of the most important aspects.

Raymarching We perform straightforward raymarching using one ray per pixel (Fig. 6). Invisible bricks are skipped (see below). For visible bricks intersected by a ray we determine their required LOD. For our tests, we choose the LOD based on the distance of a brick's center to the camera such that one voxel maps to approximately one pixel on the screen. If the desired LOD is not available in the cache, it is requested and it will be decompressed for the next frame (purple, Fig. 6). We allow this lag of one frame as the resulting artifacts are typically small when the frame rate is reasonably high. This also allows us to focus our tests on the compression scheme as the core of our method, but of course more elaborate schemes to access the desired LOD within the same frame can be used (e.g. akin to [22]). Note that rendering can always fall back to the coarsest LOD of a brick if it has not yet been decompressed, as this level is stored as the first palette entry and thus directly accessible. Bricks are flagged upon

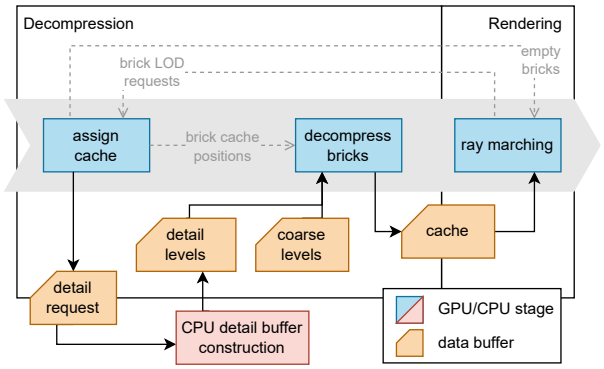

Fig. 5: An overview of our renderer: Using raymarching we read voxel-data from bricks stored in a cache. For intersected bricks that are not present in the cache, or not at the required level of detail, we generate a request. These bricks are assigned and decompressed to a free cache location, and can be accessed in the next frame. If possible, the compressed data completely resides in GPU memory. If a CSV is too large, the finest levels of detail are streamed from CPU memory to the GPU on demand.

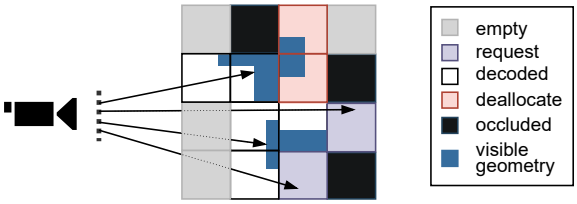

Fig. 6: Our raymarching is kept simple to focus the evaluation on the decompression performance: We march along one ray per pixel (plus one shadow ray) and perform empty-space skipping on a brick level. When the raymarcher tries to access a brick that is not present in the cache, or present but at the wrong level of detail, the brick is requested for the next frame. Bricks that are not accessed in the current frame are simply evicted from the cache.

sampling during raymarching so that bricks that become invisible can be deallocated (red, Fig. 6) in the next frame. To that end, we reset a buffer of one integer per brick to an *invisible* flag before each frame and let all ray marching threads write requested LODs in parallel. Since a requested LOD depends on the brick's center only, these are identical between threads and no race conditions occur. For hit voxels above a user-defined opacity threshold we send an additional shadow ray. Optionally we can approximate ambient occlusion by accumulation shadow rays over time (see Fig. 1, left).

Empty-Space Skipping Recall that for every compressed brick we store the palette of labels; this palette has orders of magnitude fewer entries than there are voxels in the brick. If none of the palette entries is mapped to an opacity greater than 0, then the entire brick can be skipped during raymarching and also does not need to be decompressed as long as the TF does not change its visibility.

Decompression and Caching The decompression is executed in parallel for all required bricks and their LODs directly on the GPU, and the result is put into a cache in GPU memory. The CSVs typically also reside in GPU memory, but in order to be able to render very large CSVs we optionally keep the compressed data for decoding the finest level(s) of detail in main memory and transfer it to the GPU on demand. Note that the fine LODs consume a significant portion of the data, but are only required for regions of the CSV which are visible and close to the camera.

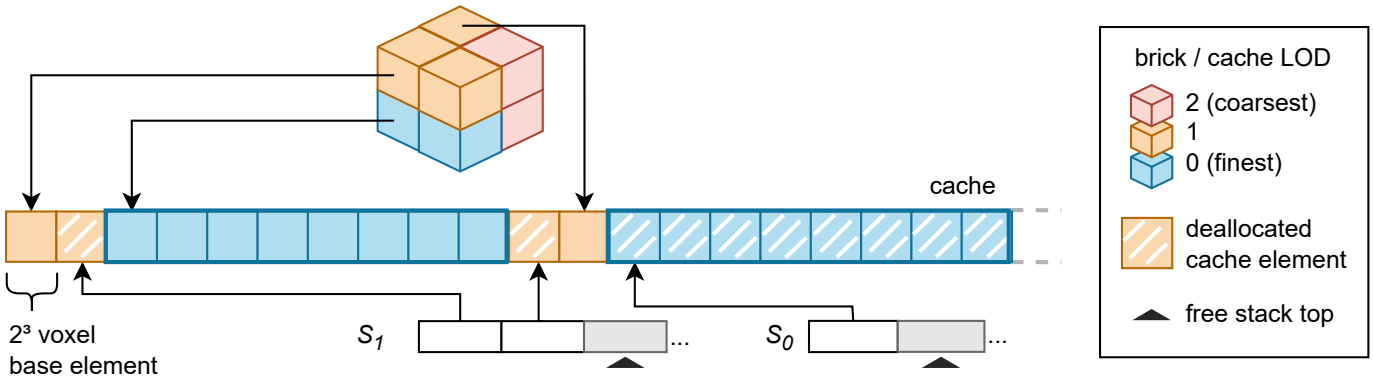

Fig. 7: Our cache is organized as base elements of 2^3 voxels which are combined to form larger blocks. In this example, bricks of LOD 1 correspond to 1 base element (2^3 voxels), LOD 0-bricks to 8 base elements (4^3 voxels). To quickly assign free locations to bricks during decompression, we use one stack S_l per LOD where one stack element points to the first base element where a block of the respective size can be stored. The locations of bricks evicted from the cache are pushed onto these stacks. Note that the coarsest LOD (here level 2) is never stored in the cache as it is available as the first entry of a brick's palette.

4.1 Brick Caching and Decoding

Our cache is loosely inspired by Shading Atlas Streaming [39] (SAS), which has been designed to stream 2D texture tiles from a powerful host to a tethered VR device. We adapt a subset of SAS to cache decompressed CSV bricks on a single GPU. In particular, we adopt the handling of free cache elements with one stack per LOD from SAS, and their usage of atomic counters to assign bricks to stack elements. However, as our LOD scheme does not create anisotropic bricks, we omit the superblock and column concepts from SAS. Instead, we simply map contiguous base elements to bricks.

For this our cache maintains a pool of base elements of 2^3 voxels which are allocated and combined to form *blocks* of sizes $2^{3l}, l = 1..N$ which can then store one decompressed brick of the respective size (Fig. 7). Bricks at the coarsest resolution $l = 0$ do not need to be decompressed. As in SAS, the index of the first base element of an already formed but currently unused block is stored on stacks S_l (one stack per block size) in order to quickly retrieve free blocks when decompressing bricks. If a stack S_l runs empty, the cache allocates new blocks of size 2^{3l} by atomically advancing a shared cache top pointer by l. After a block has been assigned to a level l, it cannot be used for other levels later. If no free base elements are available, we resort to a rebuild of the cache as proposed by SAS to remove fragmentation. Note that for reasonably sized cache pools this happens extremely seldom.

The use of the cache during visualization is as follows: After rendering a frame, the required bricks and their LODs for the next frame are known. They will then be decompressed and stored into the cache (*assign cache* in Fig. 5). For this, we first test whether a requested brick is visible for the current TF, and if visible, we mark this brick by assigning an ascending index (one sequence per block size 2^l as proposed by SAS) to it. By this, we also count how many bricks of which LOD need to be decompressed. Bricks already in the cache that became invisible because of the TF or not being hit by a ray are evicted from the cache and their blocks are pushed onto the respective stack S_l (Fig. 7).

Next we determine for each S_l if it contains enough free blocks to fulfill all requests of level l. If this is not the case, the cache allocates the new blocks required to decompress all requested bricks. As mentioned before, if not enough blocks can be allocated, the cache is rebuild, i.e. all blocks and stacks are cleared and all visible bricks of the CSV are newly decompressed. Lastly, the actual decompression of each requested brick (*decompress bricks* in Fig. 5) is carried out in parallel.

The architecture of our cache follows that of SAS with the following differences: 1) we test bricks for empty space using their palettes before requesting free locations, 2) we avoid superblocks by providing contiguous base elements using the method from above, and 3) during assignment of locations we decompress the bricks.

4.2 Detail Separation

If possible we store the CSVs in GPU memory. For too large CSVs we split the compressed data: the palette and data for the coarse levels of detail are kept on the GPU, and data for further decompressing fine detail levels is stored in CPU memory. While the latter needs a comparatively large amount of memory, bricks with high (or full) detail are often only required for close up views. The CORTEX data set (see Fig. 1), for example, has a resolution of $6144 \times 9216 \times 4096$ voxels, i.e. even at high rendering resolutions only the front most bricks might be needed with high detail. On the other hand, coarser levels are frequently accessed, also during decompressing finer levels.

To this end, we stream the detail levels for bricks to the GPU only when required (Fig. 5, bottom). For this, the cache assignment stage (Section 4.1) determines for every brick if detail levels need to be accessed for decompression. In this case, the brick index is added to a *request buffer* which is transferred to CPU memory after the assignment. A buffer containing the requested detail levels is generated in parallel to rendering the next frame and asynchronously uploaded to GPU memory. We implemented two options to handle this resulting additional lag of one frame: first we can simply decompress a brick only up to the finest available level in GPU memory, or second, we predictively upload detail information for bricks close to the camera. Once a brick has been decompressed and stored in the cache, the data for fine details is not required anymore and can be discarded. Note that in practice, even for the largest data sets tested we were able to store all but the data for decompressing level L_0 in GPU memory. For our compression of CORTEX, the total memory consumption for L_0 is 9.2 GB while all other levels take up only 4.3 GB. In our experiments we use an 8 MB buffer to upload requested detail level data to the GPU which was sufficient in most of the frames. If more detail data is requested (or the buffer would be smaller) the upload is distributed across several frames trading LOD adaptation for responsivity/interactivity.

4.3 Possible Optimizations

Here we briefly mention possible improvements for future work. There are plenty of performance optimizations known for large-volume visualization which can be combined with CSVs. For example, empty space skipping could use an octree where leaf nodes represent CSV-bricks to efficiently detect larger invisible regions during raymarching. It is also obvious that the caching strategy can be refined, e.g. by predicting bricks becoming visible within the next frames when they move towards the view frustum. Noteworthy is also that there is a large body of work on occlusion culling, either view-dependent or computed globally [18], which can be used to reduce cache usage and increase rendering performance.

Table 1: Compression rates and CPU compression times of different data sets and brick sizes, without and with using rANS-entropy coding. The third column shows the figures for our default: using rANS with two global frequency tables, one for $L_N..L_1$ and one for L_0. Timings are measured excluding data set input/output operations. For larger brick sizes we use 8 instead of 16 threads (marked by $()^+$ after the timing value).

	b	**no rANS** CR	Time (s)	GB/s	**rANS, one frequency table** CR	Time (s)	GB/s	**rANS, two frequency tables** CR	Time (s)	GB/s	data set characteristics
CELLS	16	6.958%	2.008	1.992	3.980%	2.104	1.901	3.561%	2.090	1.914	$1000 \times 1000 \times 1000$ voxels
	32	6.581%	1.859^+	2.152	3.460%	1.913	2.022	3.016%	1.957	2.043	1,067,198 labels
	64	6.428%	2.000^+	2.000	3.250%	2.189^+	1.827	**2.805%**	2.161^+	1.851	4.0 GB uncompressed
FIBER	16	3.220%	3.579	1.591	1.950%	3.812	1.494	1.737%	3.887	1.465	$1579 \times 1092 \times 1651$ voxels
	32	2.597%	3.017	1.887	1.257%	3.148	1.809	1.017%	3.744	1.521	31,877 labels
	64	2.502%	3.457^+	1.647	1.147%	3.601^+	1.581	**0.899%**	3.219^+	1.769	5.7GB uncompressed
CORTEX	16	3.684%	284.475	3.261	2.035%	360.136	2.575	1.882%	362.248	2.561	$6144 \times 9216 \times 4096$ voxels
	32	3.357%	294.042^+	3.155	1.699%	328.956^+	2.820	1.533%	320.558	2.894	15,030,572 labels
	64	3.307%	321.660^+	2.884	1.631%	351.806^+	2.637	**1.459%**	347.999^+	2.666	927.7GB uncompressed

5 RESULTS

In this section we evaluate our CSVs, compare our compression to previous work (Section 5.1, and discuss the rendering performance in Section 5.2). We will make the source code of our compression technique available. For the evaluation we use the following segmentation volumes (see Fig. 1) taken from simulations and measurements:

- CELLS: A Cellular Potts Model cancer growth simulation [44] with a resolution of $1000 \times 1000 \times 1000$ voxels and 1,067,198 labels (1067 labels/million voxels). As each individual biological cell in such simulations has its own label, this data set has by far the most labels per voxel in our evaluation.
- FIBER: A fiber segmentation of an X-ray scan of a glass fiber reinforced polymer [37] with $1579 \times 1092 \times 1651$ voxels and 31877 labels (11 labels/million voxels). Due to the low number of labels, this data set has been provided with 16 bit per voxel as opposed to the other 32 bit data sets in this evaluation. The segmentation volume contains highly anisotropic label regions and inhomogeneously shaped empty space that leads to a partial visibility for many bricks during rendering.
- CORTEX: A segmentation of an electron microscopy scan of a mouse cortex [38] with $6144 \times 9216 \times 4096$ voxels and 15,030,572 labels (65 labels/million voxels). This is the largest data set is in our evaluation with an uncompressed size of almost 928 GB and label regions of strongly varying size and shape. The resulting CSV requires storing detail level data for L_0 on the CPU. If a brick is required in full detail and L_0 has not yet been uploaded to GPU memory, the decompression temporarily uses L_1 (see Section 4.2).

5.1 Compression Performance

In this section we evaluate the compression rate for a variety of settings and compare our method to previous work. We measure time and throughput for the compression and define the reported compression rate (CR) as the ratio of compressed size to input size, i.e. smaller values mean better compression.

Parameters for CSV-compression We measure all timings on an AMD Ryzen 7 5800x 8-core CPU with 64 GB of RAM. Data sets that do not fit into RAM at once, e.g. the CORTEX, can trivially be processed in an out-of-core fashion as the (de)compression is performed for individual bricks. Note that we used a naive parallelization throughout our experiments: T threads simply compress T bricks in parallel, followed by a synchronization and concatenation of the output. In particular for larger brick sizes, this results in less optimal utilization as the processing times between threads diverge. We leave more elaborate schemes, e.g. using work queues, as well as a GPU implementation for future work.

Table 1 summarizes the results for the data sets and shows results for different brick sizes b as well as compression with and without rANS entropy coding; we use 16 threads for smaller and 8 threads for larger brick sizes. The results show that larger brick sizes (with proportionally fewer edge voxels) lead to better compression rates. This is mainly because neighbor references $R_{\{x,y,z\}}$ cannot be used across the bricks' borders. Depending on the data set, large homogeneous regions that span multiple bricks are more compactly encoded by with fewer large bricks. The rANS coding improves the overall CRs by about 40% compared to storing 4 bit nibbles directly. As mentioned before, for the compression with rANS we compute two global frequency tables per data set—one for nodes in L_0 where the stop bit is always 0, and one for all other nodes—during an encoding prepass using every 512^{th} brick, except for CORTEX where we use every 4096^{th} brick (Section 3.2). In our experiments, CRs only varied very subtly when increasing or decreasing the subsampling. We observe compression speeds of 1.5 to 3 GB/s for all data sets; compression with $b = 32$ consistently achieves the highest throughput, despite the lower utilization of CPU cores. rANS coding only slightly impact the throughput by about 2.5% on average.

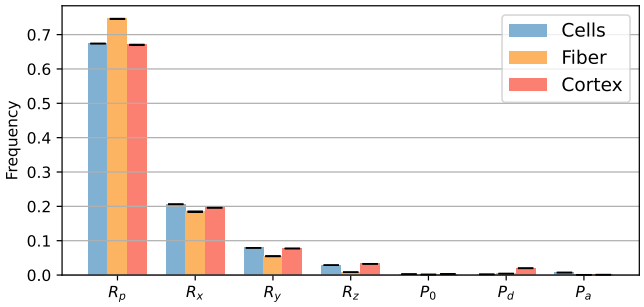

Fig. 8: Mean relative frequencies of operations for the CELLS, FIBER, and CORTEX data sets. We also show the standard deviation σ. Note that we did not compute it from the bricks' operation counts directly, as some bricks can be encoded with very few operations and would lead to large σ without informative value. Instead we compute the relative frequencies 2048 times for 100 randomly chosen bricks ($b = 16$), and obtain the standard deviation from these frequencies.

Comparison to other Methods We compare the compression rate (CR) and time to the following methods (Tab. 2):

- hdf5: Segmented volumes are often provided in hdf5/Hierarchical Data Format [38] which uses a brick-wise gzip-compression (LZ77 with Huffman coding); the brick size typically is 128^3.
- png: Another common way is to slice volumes and store image stacks in Portable Network Graphics (PNG)-format where labels are split into 8-bit RGBA channels [3]. For our comparison, we

Table 2: Compression rates of different techniques: hdf5 uses gzip on bricks of 128^3 voxels, for png we sliced the volume and used zlib with the highest compression level. For Compresso we used the default parameters with a window size of (8,8,1). For Neuroglancer we set the block size to (8,8,8). Our technique (with $b = 64$) is benchmarked without and with rANS using two frequency tables to assess the overhead of entropy coding. Timings are reported for single-threaded execution; the timings for our method running with 16 threads are shown in parentheses. Note that the full CORTEX data set did not fit into memory and the available implementations of Compresso and Neuroglancer did not handle out-of-core compression. To this end, we chose a representative 1024^3 subvolume of CORTEX which matches the overall average number of labels per voxel and whose hdf5-compression ratio is the same as for the entire CORTEX data.

data set	size/#labels	hdf5	png	Compresso	Compresso+LZMA	Neuroglancer	ours (no rANS)	ours (rANS)
CELLS	4.0GB	7.221%	10.812%	8.337%	2.753%	13.622%	6.428%	2.805%
	1M labels	23s	493s	35s	135s	11s	15s (3s)	16s (3s)
FIBER	5.7GB	3.051%	3.665%	26.700%	5.861%	3.658%	2.502%	0.899%
	32K labels	53s	380s	130s	619s	12s	19s (6s)	20s (6s)
CORTEX*	4.3GB	2.459%	2.406%	8.515%	1.267%	3.999%	3.564%	1.590%
	45K labels	18s	138s	35s	138s	5s	10s (3s)	11s (3s)

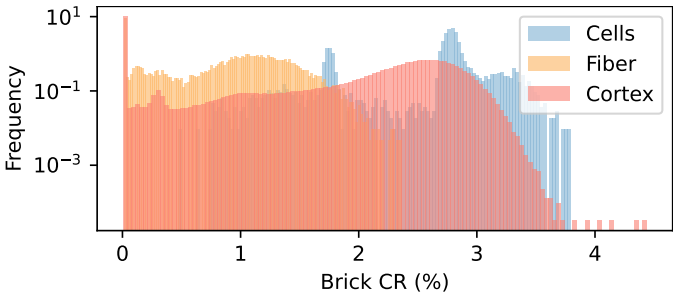

Fig. 9: Log-scale histogram of compression rates per brick with $b = 64$. We observe that even the worst case bricks still have compression rates of 3.794% (CELLS), 2.360% (FIBER), and 4.436% (CORTEX). Note that the peak near zero for CORTEX and FIBER is due to completely homogeneous bricks represented as a single P_a entry.

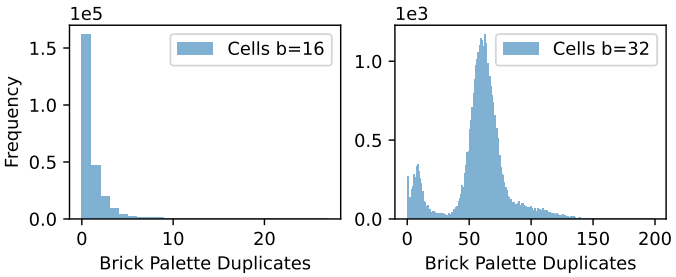

Fig. 10: Histogram of the number of duplicate label entries in brick palettes for CELLS. On average a brick for $b = 16$ has only 0.8 duplicate entries. Choosing $b = 32$ results in 37449 bricks which contain 58 duplicates on average. This number remains relatively low also thanks to the P_δ-operations. The FIBER and CORTEX data sets have significantly fewer different labels per brick and duplicates are rare.

slice the volumes along their z-axes and compress each 2D slice with the highest zlib compression level 9.

- Compresso [36]: This method has been designed for segmentation volumes and its CR often outperforms other techniques. We compare to the improved Compresso version 3.2.

- Neuroglancer [20]: A web-based volume viewer with a GPU-friendly compression format that stores a palette for each brick which is then indexed by the voxels.

We can make the following observations: Compared to hdf5 our CSVs achieve better compression although we use smaller brick sizes. Without LZMA, Compresso consistently yields worse CRs than our method without rANS. Compresso with LZMA achieves roughly similar CRs as our method for CELLS and CORTEX. The result for the FIBER-volume is significantly worse than all other methods, which presumably is due to the strong anisotropy of the features and bad matching of representative windows used for compression in Compresso. As an experiment we also used a global LZMA compression (as Compresso does) with the output of our brick encoding. By this we achieve slightly better CRs than Compresso (e.g. 2.579% for ours and 2.753% for Compresso on CELLS). Note, however, that Compresso itself as well as using a single LZMA stream make level-of-detail rendering and brick-wise decompression impossible. This is why Compresso and image stack approaches are not directly applicable for volume rendering. hdf5-volumes, in contrast, can be decompressed per brick, but also have no elaborate level-of-detail mechanism. Neuroglancer [20] is meant for direct volume rendering and can also be trivially extended to store multiple levels-of-detail. However, already without LODs the compression rates are worse than with all other techniques.

As we can see, our CSV are competitive with the CRs of the state-of-the-art compression methods for segmentation volumes, and at the same time they can be directly used with volume rendering due to supporting brick-wise decompression and adaptive level-of-detail.

Tab. 2 shows *single-threaded* compression time for all methods as multi-threading implementations were not available for all methods. While the simple Neuroglancer-method encodes about twice as fast as our CSVs, its CR is significantly worse. Note that our CSVs are $8-30\times$ faster than Compresso with LZMA while they achieve similar CRs (and significantly better CR for FIBER). Our rANS encoding introduces only little overhead.

Encoding Operation Frequencies Fig. 8 shows the relative frequencies of the encoding operations; note that they vary slightly, but not fundamentally, over the data sets. We further observe that 95% of the used operations reference other nodes ($R_p, R_{\{x,y,z\}}$), and only 0.08% (FIBER) to 0.7% (CELLS) of nodes use the costly P_a operation, adding an entry to the palette. Recall that the frequencies of operations are also influenced by the order in which they are tested during encoding. For example, $R_{\{x,y,z\}}$ would be more evenly distributed if tested in random order (which would harm compression).

As mentioned before we obtain the frequencies for encoding by sub-sampling bricks in a prepass. The cost for the prepass depends linearly on the number of sampled bricks and thus yields a significant speedup. The compression rate with sub-sampled frequencies compared to a full pass over all bricks differs only on the fifth decimal digit, i.e. the frequencies vary only minimally over the bricks of the data set we tested. In applications where a prepass is not practical, static (predetermined) frequency tables would still result in good compression ratios and they could still be tailored, for example, to a specific application (Cellular Potts Model, electron microscopy etc.).

In our tests, we observed data sets with bricks which showed noticeable worse compression ratio than the average – however, the CR still remained below 5%, which is less than the simple paletting as used by Neuroglancer (Fig. 9). We found that such bricks contain segmentation

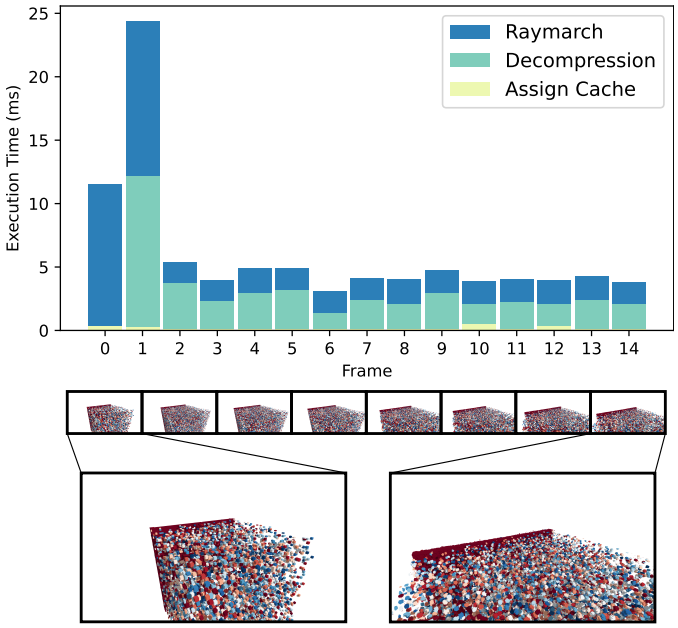

Fig. 11: A stress test for the decompression with the CELLS data set: a cold start, fast moving camera, and bricks becoming visible. We show the times for raymarching, decompression, and cache assignment at 1920×1080 resolution. The bottom row shows the individual frames (note how the coarsest LOD is used in frame 0 as no bricks are stored in the fresh cache). Changing brick LODs and changes in visibility also create decompression workload in subsequent frames.

or simulation errors resulting in many disconnected label regions which inevitably lead to palette duplicates. Fig. 10 shows histograms for duplicate palette entries for CELLS (FIBER and CORTEX exhibit almost no duplicates). Smaller brick sizes lead to fewer duplicates which is to be expected; larger bricks often span differently labelled regions and the required palette index of a node during encoding might not be adjacent and out of range for P_δ operations.

5.2 Rendering Performance

We evaluate our raymarcher with the same hardware configuration as above using an NVIDIA RTX 3070 Ti with 8 GB of video memory. We use a cache size for decompressed bricks of 1 GB for CELLS and FIBER; for these we can easily store the full CSV in video memory. For CORTEX we set the cache size to 2 GB and separate the detail levels L_0 which consumes 9.2GB of CPU memory for all bricks. Recall that this detail is streamed to the GPU on-demand; the levels $L_l, l > 0$ are kept in GPU memory and require 4.3 GB. Table 3 shows rendering performance with minimum, average, and maximum milliseconds (ms) per frame rendered at a resolution of 1920×1080, and also lists the decompression throughput on the GPU. Table 4 shows rendering performance using different brick sizes for CELLS and FIBER. Smaller brick sizes generally lead to slightly faster render times. If not stated otherwise, we use $b = 32$ as a compromise between rendering performance and compression, except for CORTEX where we use $b = 64$ for better compression and to minimize GPU memory usage. As expected, the on-demand streaming of detail levels L_0 from CPU to GPU memory results in a slight reduction of rendering performance, but its influence is only notably evident for the maximum frame times. The maximum frame time for the large CORTEX data set (172 ms) is due to a full cache rebuild (also see the supplemental video). The average render times as well as the maximum times for CELLS and FIBER are only slightly affected by detail streaming. This is because in both cases all decompressed bricks easily fit into the cache. For FIBER usually up to 600 MB of the cache are used at any point, while CELLS requires less than 200 MB of the cache for decoded bricks. With CORTEX, we experience the cache quickly filling up to its 2 GB limit in close camera

Table 3: Average, minimum, and maximum total rendering times per frame in milliseconds (ms) for a fly-around at 1920×1080 resolution, and GPU-decompression performance in GB/s for our data sets. The render times include one shadow ray per pixel. First column: rendering with CSVs completely stored in GPU-memory; second column: CSV split such that L_0 is stored in CPU-memory.

	CSV on GPU min / avg / max	L_0 on CPU min / avg / max	decoding in GB/s
CELLS	5 / 17 / 29	5 / 20 / 39	9.9
FIBER	7 / 18 / 23	7 / 22 / 30	10.4
CORTEX	–	7 / 40 / 172	9.3

Table 4: Average and maximum frame times in milliseconds with varying brick sizes b for a fly through over CELLS and FIBER. CORTEX cannot be rendered with $b < 64$ on our GPU with 8 GB memory.

	CSV on GPU			L_0 on CPU		
b	16	32	64	16	32	64
CELLS	6 / 12	17 / 29	23 / 37	9 / 15	20 / 39	24 / 43
FIBER	10 / 13	18 / 23	33 / 39	12 / 15	22 / 30	35 / 39

views when using transfer functions that lead to many visible bricks in the finest LOD. We leave more efficient caching schemes for future work.

Fig. 11 shows the total frame times as well as the portions spent for raymarching, decoding, and cache assignment for a synthetic test using CELLS designed to put stress on the decompression by fast camera movement and change in the visibility of bricks. The short sequence consists of 15 frames where the cache is empty in the beginning. Consequently frame 0 is rendered using the coarsest LOD only and significant decompression load is generated for frame 1. Even the initial decompression of all bricks visible in frame 0 results in a total frame time below 25 ms. Note that similar cases in practice only occur once at the beginning of rendering a sequence or when the camera view changes (almost) completely. The subsequent frames in the experiment still require further decompression due to bricks becoming visible and selection of LODs, but yield roughly equal total frame times below 5ms. Also note that the cache assignment stage, which among others checks a brick's visibility by applying the transfer function to the palette, only accounts for a negligible overhead.

6 CONCLUSIONS

We presented a novel lossless compression technique for voxel-based segmentation volumes which achieves compression ratios comparable to, or better than the state-of-the-art. At the same time, its brick-wise compression provides sufficient granularity for efficient volume visualization, and the multi-resolution encoding inherently enables decompression with adaptive level-of-detail. We have demonstrated volume visualization using raymarching and caching of decompressed bricks for data sets with more than 900 GB on modest hardware at real-time frame rates. We have also outlined possible venues for future work to further improve the performance by increasing the utilization of CPU cores, improved empty-space skipping, predictive decompression, or caching. Even without these optimizations our method achieves high throughput for the compression and real-time visualization of very large segmentation volumes.

SUPPLEMENTAL MATERIALS

Please see (1) the accompanying video showcasing our technique and (2) the source code of our compression method released under a CC BY-NC 4.0 license. We also provide an (3) in-depth comparison of using Morton over Hilbert curves in the compression.

Acknowledgments

The authors wish to thank the NIC Research Group Computational Structural Biology at Jülich Research Center for simulating and providing the Cellular Potts Model data sets and for their helpful feedback. We also wish to thank the Research Group Computed Tomography, University of Applied Sciences Upper Austria, Campus Wels, where the Fiber data was measured and analyzed for providing this data set. This work has been supported by the Helmholtz Association (HGF) under the joint research school "HIDSS4Health – Helmholtz Information and Data Science School for Health" and through the Pilot Program Core Informatics.

References

[1] M. Agus, A. Aboulhassan, K. Al Thelaya, G. Pintore, E. Gobbetti, C. Calì, and J. Schneider. Volume Puzzle: visual analysis of segmented volume data with multivariate attributes. In *Proc. IEEE Visualization and Visual Analytics*, pp. 130–134. IEEE, Los Alamitos, 2022. doi: 10.1109/VIS54862.2022.00035 1, 2

[2] A. K. Ai-Awami, J. Beyer, D. Haehn, N. Kasthuri, J. W. Lichtman, H. Pfister, and M. Hadwiger. NeuroBlocks – Visual Tracking of Segmentation and Proofreading for Large Connectomics Projects. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):738–746, 2016. doi: 10.1109/TVCG.2015.2467441 2

[3] K. Al-Thelaya, M. Agus, and J. Schneider. The Mixture Graph – A Data Structure for Compressing, Rendering, and Querying Segmentation Histograms. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):645–655, 2021. doi: 10.1109/TVCG.2020.3030451 2, 7

[4] K. Anagnostou, T. J. Atherton, and A. E. Waterfall. 4D Volume Rendering with the Shear Warp Factorisation. In *Proc. IEEE Symposium on Volume Visualization*, p. 129–137. ACM, New York, 2000. doi: 10.1145/353888.353909 2

[5] M. Balsa Rodríguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum*, 33(6):77–100, 2014. doi: 10.1111/cgf.12280 1, 2

[6] D. R. Berger, H. S. Seung, and J. W. Lichtman. VAST (Volume Annotation and Segmentation Tool): Efficient Manual and Semi-Automatic Labeling of Large 3D Image Stacks. *Frontiers in Neural Circuits*, 12, 2018. doi: 10.3389/fncir.2018.00088 2

[7] M. Berghoff, J. Rosenbauer, F. Hoffmann, and A. Schug. Cells in Silico – introducing a high-performance framework for large-scale tissue modeling. *BMC Bioinformatics*, 21(436):1–21, 2020. doi: 10.1186/s12859-020-03728-7 1

[8] J. Beyer, A. Al-Awami, N. Kasthuri, J. W. Lichtman, H. Pfister, and M. Hadwiger. ConnectomeExplorer: Query-Guided Visual Analysis of Large Volumetric Neuroscience Data. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2868–2877, 2013. doi: 10.1109/TVCG.2013.142 2

[9] J. Beyer, M. Hadwiger, A. Al-Awami, W.-K. Jeong, N. Kasthuri, J. W. Lichtman, and H. Pfister. Exploring the Connectome: Petascale Volume Visualization of Microscopy Data Streams. *IEEE Computer Graphics and Applications*, 33(4):50–61, 2013. doi: 10.1109/MCG.2013.55 2

[10] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum*, 34(8):13–37, 2015. doi: 10.1111/cgf.12605 2

[11] J. Beyer, H. Mohammed, M. Agus, A. K. Al-Awami, H. Pfister, and M. Hadwiger. Culling for Extreme-Scale Segmentation Volumes: A Hybrid Deterministic and Probabilistic Approach. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Scientific Visualization)*, 25(1), 2018. doi: 10.1109/TVCG.2018.2864847 2

[12] J. Beyer, J. Troidl, S. Boorboor, M. Hadwiger, A. Kaufman, and H. Pfister. A Survey of Visualization and Analysis in High-Resolution Connectomics. *Computer Graphics Forum*, 41(3):573–607, 2022. doi: 10.1111/cgf.14574 1

[13] V. Careil, M. Billeter, and E. Eisemann. Interactively Modifying Compressed Sparse Voxel Representations. *Computer Graphics Forum*, 39(2):111–119, 2020. doi: 10.1111/cgf.13916 2

[14] J. Choi, D. G. C. Hildebrand, J. Moon, T. M. Quan, T. A. Tuan, S. Ko, and W.-K. Jeong. ZeVis: A Visual Analytics System for Exploration of a Larval Zebrafish Brain in Serial-Section Electron Microscopy Images. *IEEE Access*, 9:78755–78763, 2021. doi: 10.1109/ACCESS.2021.3084066 2

[15] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann. Geometry and Attribute Compression for Voxel Scenes. *Computer Graphics Forum*, 35(2):397–407, 2016. doi: 10.1111/cgf.12841 2

[16] D. Dolonius, E. Sintorn, V. Kämpe, and U. Assarsson. Compressing Color Data for Voxelized Surface Geometry. *IEEE Transactions on Visualization and Computer Graphics*, 25(2):1270–1282, 2019. doi: 10.1109/TVCG.2017.2741480 2

[17] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *Picture Coding Symposium*, pp. 65–69. IEEE, Los Alamitos, 2015. doi: 10.1109/PCS.2015.7170048 2, 4

[18] M. Ernst, F. Firsching, and R. Grosso. Entkerner: A System for Removal of Globally Invisible Triangles from Large Meshes. In *Proc. of the International Meshing Roundtable*, pp. 449–458. Sandia National Laboratories, Albuquerque, 2004. 6

[19] J. E. Fowler and R. Yagel. Lossless Compression of Volume Data. In *Proc. IEEE Symposium on Volume Visualization*, p. 43–50. ACM, New York, 1994. doi: 10.1145/197938.197961 2

[20] Google Inc. Neuroglancer. https://github.com/google/neuroglancer, 2016. 2, 8

[21] S. Guthe and M. Goesele. GPU-based lossless volume data compression. In *3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video*, pp. 1–4. IEEE, Los Alamitos, 2016. doi: 10.1109/3DTV.2016.7548892 2

[22] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974–983, 2018. doi: 10.1109/TVCG.2017.2744238 5

[23] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012. doi: 10.1109/TVCG.2012.240 2

[24] A. Hussain, A. Al-Fayadh, and N. Radi. Image compression techniques: A survey in lossless and lossy algorithms. *Neurocomputing*, 300:44–69, 2018. doi: 10.1016/j.neucom.2018.02.094 2

[25] I. Ihm and S. Park. Wavelet-Based 3D Compression Scheme for Interactive Visualization of Very Large Volume Data. *Computer Graphics Forum*, 18(1):3–15, 1999. doi: 10.1111/1467-8659.00298 2

[26] D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski. A Survey of Volumetric Illumination Techniques for Interactive Volume Rendering. *Computer Graphics Forum*, 33(1):27–51, 2014. doi: 10.1111/cgf.12252 5

[27] V. Kämpe, E. Sintorn, and U. Assarsson. High Resolution Sparse Voxel DAGs. *ACM Transactions on Graphics*, 32(4):101:1–101:12, 2013. doi: 10.1145/2461912.2462024 2

[28] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature Sensitive Surface Extraction from Volume Data. In *Proc. ACM SIGGRAPH*, p. 57–66. ACM, New York, 2001. doi: 10.1145/383259.383265 2

[29] S. Laine and T. Karras. Efficient Sparse Voxel Octrees. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, p. 55–63. ACM, New York, 2010. doi: 10.1145/1730804.1730814 2

[30] V. Lempitsky. Surface extraction from binary volumes with higher-order smoothness. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1197–1204. IEEE, Los Alamitos, 2010. doi: 10.1109/CVPR.2010.5539832 2

[31] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006. doi: 10.1109/TVCG.2006.143 2

[32] Y. Lu, K. Jiang, J. A. Levine, and M. Berger. Compressive neural representations of volumetric scalar fields. *Computer Graphics Forum*, 40(3):135–146, 2021. doi: 10.1111/cgf.14295 2

[33] B. Madoš, E. Chovancová, and M. Hasin. Evaluation of pointerless sparse voxel octrees encoding schemes using huffman encoding for dense volume datasets storage. In *International Conference on Emerging eLearning Technologies and Applications*, pp. 424–430. IEEE, Piscataway, 2020. doi: 10.1109/ICETA51985.2020.9379265 2

[34] B. Madoš and N. Ádám. Evaluation of Encoding Schemas for Optimization of Bit-Level Run-Length Encoding Within Lossless Compression of Binary Images. In *IEEE International Conference on Intelligent Engineering Systems (INES)*, pp. 75–80. IEEE, Los Alamitos, 2019. doi: 10.1109/INES46365.2019.9109528 2

[35] B. Madoš, N. Ádám, and M. Štancel. Representation of Dense Volume Datasets Using Pointerless Sparse Voxel Octrees With Variable and

Fixed-Length Encoding. In *IEEE World Symposium on Applied Machine Intelligence and Informatics*, pp. 000343–000348. IEEE, Los Alamitos, 2021. doi: 10.1109/SAMI50585.2021.9378675 2

[36] B. Matejek, D. Haehn, F. Lekschas, M. Mitzenmacher, and H. Pfister. Compresso: Efficient Compression of Segmentation Data For Connectomics. In *Medical Image Computing and Computer-Assisted Intervention*, pp. 781–788. Springer, Cham, 2017. doi: 10.1007/978-3-319-66182-7_89 2, 8

[37] J. Maurer, D. Salaberger, M. Jerabek, J. Kastner, and Z. Major. Quantitative investigation of local strain and defect formation in short glass fibre reinforced polymers using X-ray computed tomography. *Nondestructive Testing and Evaluation*, 37(5):582–600, 2022. doi: 10.1080/10589759.2022.2075865 7

[38] A. Motta, M. Berning, K. M. Boergens, B. Staffler, M. Beining, S. Loomba, P. Hennig, H. Wissler, and M. Helmstaedter. Dense connectomic reconstruction in layer 4 of the somatosensory cortex. *Science*, 366(6469):eaay3134, 2019. doi: 10.1126/science.aay3134 1, 7

[39] J. H. Mueller, P. Voglreiter, M. Dokter, T. Neff, M. Makar, M. Steinberger, and D. Schmalstieg. Shading Atlas Streaming. *ACM Transactions on Graphics*, 37(6):119:1–119:16, 2018. doi: 10.1145/3272127.3275087 2, 6

[40] K. Museth. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In *ACM SIGGRAPH Talks*, pp. 1:1–1:2. ACM, New York, 2021. doi: 10.1145/3450623.3464653 2

[41] T. S. Newman and H. Yi. A survey of the marching cubes algorithm. *Computers and Graphics*, 30(5):854–879, 2006. doi: 10.1016/j.cag.2006.07.021 2

[42] M. A. Rahman and M. Hamada. Lossless Image Compression Techniques: A State-of-the-Art Survey. *Symmetry*, 11(10), 2019. doi: 10.3390/sym11101274 2

[43] M. L. Rhodes, J. F. Quinn, and J. Silvester. Locally Optimal Run-Length Compression Applied to CT Images. *IEEE Transactions on Medical Imaging*, 4(2):84–90, 1985. doi: 10.1109/TMI.1985.4307701 2

[44] J. Rosenbauer, M. Berghoff, and A. Schug. Emerging Tumor Development by Simulating Single-cell Events. *bioRxiv*, 2020. doi: 10.1101/2020.08.24.264150 7

[45] J. Troidl, C. Cali, E. Gröller, H. Pfister, M. Hadwiger, and J. Beyer. Barrio: Customizable Spatial Neighborhood Analysis and Comparison for Nanoscale Brain Structures. *Computer Graphics Forum*, 41(3):183–194, 2022. doi: 10.1111/cgf.14532 2

[46] A. J. Villanueva, F. Marton, and E. Gobbetti. SSVDAGs: Symmetry-Aware Sparse Voxel DAGs. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, p. 7–14. ACM, New York, 2016. doi: 10.1145/2856400.2856420 2

[47] S. Weiss, P. Hermüller, and R. Westermann. Fast Neural Representations for Direct Volume Rendering. *Computer Graphics Forum*, 41(6):196–211, 2022. doi: 10.1111/cgf.14578 2

[48] A. Weißenberger and B. Schmidt. Massively Parallel Huffman Decoding on GPUs. In *Proc. International Conference on Parallel Processing*, pp. 27:1–27:10. ACM, New York, 2018. doi: 10.1145/3225058.3225076 2

[49] J. Weissenböck, A. Amirkhanov, W. Li, A. Reh, A. Amirkhanov, E. Gröller, J. Kastner, and C. Heinzl. FiberScout: An Interactive Tool for Exploring and Analyzing Fiber Reinforced Polymers. In *IEEE Pacific Visualization Symposium*, pp. 153–160. IEEE, Los Alamitos, 2014. doi: 10.1109/PacificVis.2014.52 1, 2

[50] L. Williams. Pyramidal Parametrics. In *Computer Graphics (Proc. SIGGRAPH)*, p. 1–11. ACM, New York, 1983. doi: 10.1145/800059.801126 4