


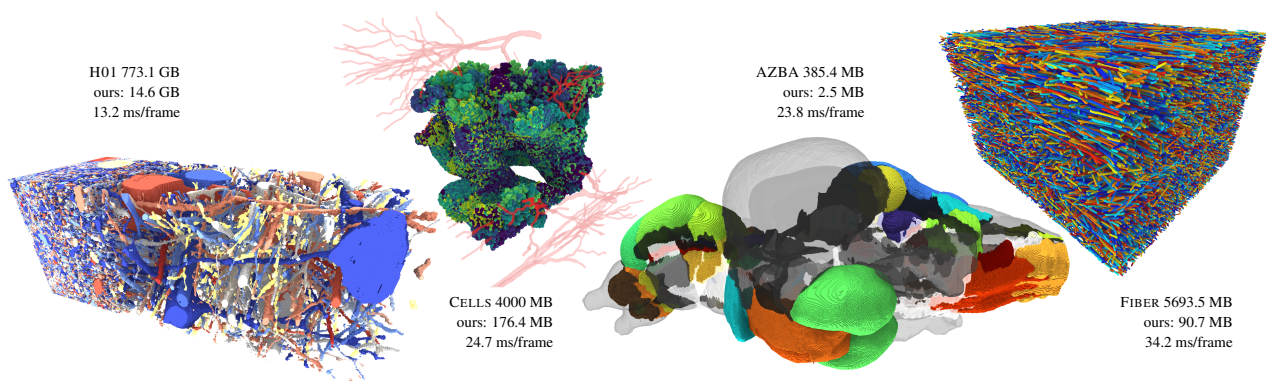


# Random Access Segmentation Volume Compression for Interactive Volume Rendering

M. Piochowiak , F. Kurpicz , and C. Dachsbacher 

Karlsruhe Institute of Technology, Germany



**Figure 1:** H01, CELLS, AZBA, and FIBER data sets compressed with our full random access multi-resolution CSGV-R segmentation volume compression. Renderings are in FullHD with a static camera and accumulate one indirect ambient occlusion ray per pixel per frame over 256 frames. The CSGV-R volumes ( $b = 64$  with stop bits) are rendered in the compression domain, accelerated with a voxel cache of 1 GiB size, plus an empty space cache except for H01. DDA ray marching accesses the multi-resolution volumes so that 1 voxel approximately maps to one pixel in the image. The renderings contain no decompression artifacts from caching or streaming as seen in other techniques.

## Abstract

Segmentation volumes are voxel data sets often used in machine learning, connectomics, and natural sciences. Their large sizes make compression indispensable for storage and processing, including GPU video memory constrained real-time visualization. Fast Compressed Segmentation Volumes (CSGV) [PD24] provide strong brick-wise compression and random access at the brick level. Voxels within a brick, however, have to be decoded serially and thus rendering requires caching of visible full bricks, consuming extra memory. Without caching, accessing voxels can have a worst-case decoding overhead of up to a full brick (typically over 32,000 voxels). We present CSGV-R which provide true multi-resolution random access on a per-voxel level. We leverage Huffman-shaped Wavelet Trees for random accesses to variable bit-length encoding and their rank operation to query label palette offsets in bricks. Our real-time segmentation volume visualization removes decoding artifacts from CSGV and renders CSGV-R volumes without caching bricks at faster render times. CSGV-R has slightly lower compression rates than CSGV, but outperforms Neuroglancer, the state-of-the-art compression technique with true random access, with  $2\times$  to  $4\times$  smaller data sets at rates between 0.648% and 4.411% of the original volume sizes.

## CCS Concepts

• **Computing methodologies** → Rendering; • **Information systems** → Data compression;

## 1. Introduction

Segmentation volumes are commonly used in various of domains such as connectomics, machine learning, or natural sciences. It is desirable to exploit the computational resources and parallel processing of GPUs when working with large-scale segmentation volumes, but the available GPU memory can quickly become a limiting factor. Working with *compressed* segmentation volumes could alleviate

this problem, however, existing compression methods are either not suitable for GPU-based processing and rendering, or have significant shortcomings, most notably not offering strong compression *and* random access at the same time.

The state-of-the-art method Fast Compressed Segmentation Volumes (CSGV) [PD24] encodes individual bricks of segmentation volumes in a multi-resolution fashion and allows random access on

this level. However, within a brick, voxels can only be decoded from a stream of stored operations in a strictly serial manner, prohibiting true random access. This has downsides, e.g. for rendering the volumes: First, a cache for decoded bricks is needed which stores many more voxels than required and consumes significant memory—up to gigabytes in addition to the compressed data. At high rendering resolutions where many bricks are requested at the finest level-of-detail, the cache size may easily exceed the available GPU memory. Due to diminishing returns of improving compression rates, we suggest that novel methods address this dependency on large caches to reduce overall rendering memory. Second, there is no feasible way of accessing a voxel if it is not in the cache. For tasks with incoherent random single voxel accesses, the decoding overhead would be significant as bricks always have to be decoded up to the requested voxel.

In this paper, we introduce a novel compressed data structure for segmentation volumes with true random access to voxels. Building on the compact multi-resolution encoding of CSGV, a key component of our Random Access Compressed Segmentation Volumes (CSGV-R) is using wavelet tree encodings for indexing the compressed data. Wavelet trees are used in text indexing and related fields, and to the best of our knowledge, we are the first to adapt and use them in a rendering context. Our data structure does not require the serial variable bit-length asymmetric numerical systems (ANS) encoding [Dud13] as CSGV, and is thus more GPU-friendly. We reduce the implementation and query overhead of our wavelet trees to surpass CSGV's render performance. Our random access removes the need for caching on the brick level: we evaluate rendering segmentation volumes without any caching, and with caching individual voxels directly. To summarize, our contributions are:

- a GPU-friendly, true random access and lossless compression method for segmentation volumes,
- an efficient renderer with optional, random access-enabled caching of individual voxels (using Cuckoo hashing), and
- an optimized Huffman-shaped wavelet matrix GPU-shader implementation for indexing compressed volume data.

## 2. Related Work and Background

In the following, we introduce related works from segmentation volume compression and visualization, as well as wavelet trees.

### 2.1. Segmentation Volume Compression

A large body of work exists on gray-scale image and volume compression [BTB\*22, BRGIG\*14]. Such compressors are applied to imaging data if it is paired with segmentation volumes [MJB\*21]. But they are rarely directly used for segmentation volumes as compression of those is usually lossless [MHL\*17, Goo16, PD24] and their content, integer classification labels, behaves different than image data. An exception are lossless image compressors like PNG [ATAS21]. Lossless general purpose compression, e.g. gzip, is used in compressed file formats like hdf5 [HDF] and NifTI [LM14]. For binary segmentations, sparse-voxel-octrees [LK10] and directed-acyclic-graphs [KSA13] can be applied. For data with more labels, label values and occupancy can be decoupled within local bounding boxes [WPD24] but compression rates are limited. Compresso [MHL\*17] encodes label regions within small windows as border bit masks and label palettes. Windows can be randomly accessed,

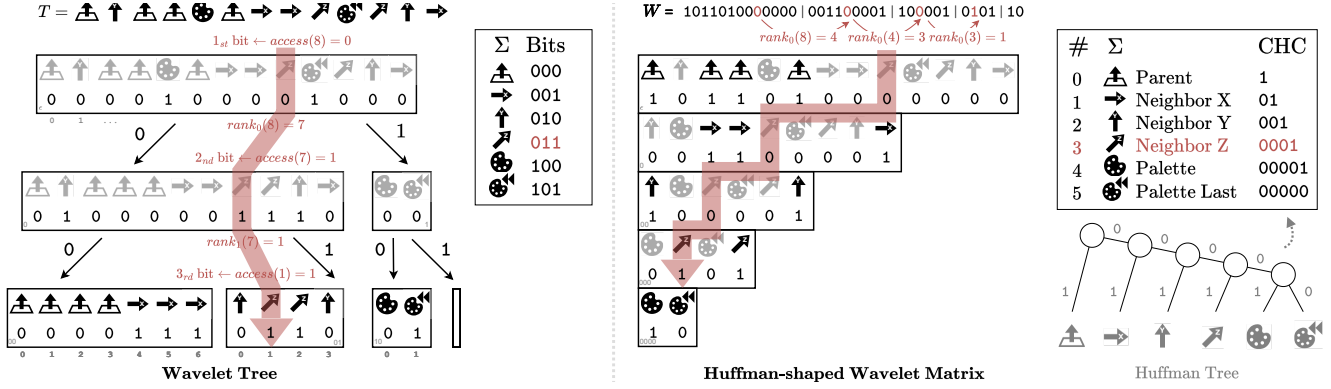
but strong compression rates are only achieved when additionally applying a global LZMA compression, which makes random access impossible and is not parallelizable. The Mixture Graph [ATAS21] creates a graph-compacted multi-resolution hierarchy of label distributions. The Google Neuroglancer precomputed format [Goo16] is a brick-wise encoding storing a palette of labels for each brick and a fixed number of bits per voxel indexing into it. This is the fastest format, offers true random access down to the voxel level, but has sub-par compression rates. CSGV [PD24] is the state-of-the-art segmentation volume compressor in terms of compression rates and operates brick-wise as well. It encodes a multi-resolution hierarchy of labels as a stream of label assignment operation. The stream is compressed with an ANS [Dud13] variable bit-length encoding. While bricks can be randomly accessed, access within a brick, covering up to several thousand voxels, must happen strictly serially. Our multi-resolution segmentation volume compressor is an alternative to CSGV that offers random access down to the voxel level as in Google Neuroglancer but with compression rates close to CSGV.

### 2.2. Segmentation Volume Visualization

Segmentations are often visualized within labeling or proofreading frameworks as 2D slices or rasterized 3D triangle meshes [PHK04, BBB\*17, BSL18, AABH\*16]. Meshing of voxel regions is either a preprocess, or only supported for few labels [Lem10, QDB11, LC87]. Portable visualization methods are often web based [Goo16, BBB\*17] while offline rendering may use photorealistic renderers [Ble18, VMM\*23] and distributed hardware [MAF07] with longer render times and lower accessibility. For large-scale segmentation volumes, distant storage formats allow partially streamed access with caching [MSL22] as implemented by Google Neuroglancer [Goo16] and Webknossos [BBB\*17] that both render 2D slices and subset 3D meshes. Ray marching can be used when voxel data is directly rendered instead of meshed representations [ATAS21, PD24]. If volumes fit into main memory, renderers on single GPU consumer hardware may stream and cache partial volumes from main memory to GPU video memory (VRAM) [BAAK\*13, BMA\*18]. Alternatively, a compressed representation of the volume is stored in VRAM to partially decode and cache visible regions in a faster, fully GPU-based pipeline [PD24]. With compression domain rendering, caching is not required [WPD24, Goo16]. Werner et al. [WPD24] encode label occupancy inside separate bounding boxes, making their compression suitable for hardware accelerated raytracing. The method requires sparsity in the data and achieves weaker compression than CSGV. The Mixture Graph [ATAS21] solves the problem of color filtering, which can only be applied with post-classification transfer functions [EHK\*06] for multi-resolution segmentation volumes. The Volume Conductor [LAB\*24] solves visual clutter in densely labeled volumes with label-based visibility management. Volume Puzzle [AAAT\*22] offers semi-automatic transfer function creation for segmentation volumes. Our renderer stores random access compressed volumes in VRAM for a fully GPU-based compression domain rendering pipeline with optional voxel caching.

### 2.3. Access and Rank Queries on Bit Vectors

A bit vector is a text  $B$  over the binary alphabet  $\{0, 1\}$ . On bit vectors, we are interested in two types of queries:  $access(i) = B[i]$



**Figure 2:** A wavelet tree that encodes a text  $T$  of operations  $\omega \in \Sigma$  in its conceptual form and as a compressed Huffman-shaped wavelet matrix. **Wavelet tree** traversal for  $\text{access}(8) = T[8] = \nearrow$  is highlighted in red: Symbols are constructed bit-by-bit from the level-wise bit vectors, starting from the top. A bit vector access yields the next bit  $\alpha$  which also determines if descending left 0 or right 1 to the next level. A  $\text{rank}_\alpha$  determines the read index in the next level. The **Huffman-shaped wavelet matrix** is a pointer-less memory layout of a wavelet tree. It stores symbols as their variable bit-length canonical Huffman-codes (CHC) to compress  $T$ . CHC are constructed from the Huffman-tree (right) in which frequent symbols have shorter lengths. Our CHC create an efficient tree layout where any 1 bit terminates a character. Thus, no right children exist and  $\text{rank}_0$  queries return the index in the next level if a 0 bit was read in the current level. Note that explicit symbols are added for readability only: only the bit vector  $W$  and the wavelet tree level start indices  $W_1 \dots W_4$  within it are stored in memory.

and  $\text{rank}_\alpha(i) = |\{j < i : B[j] = \alpha\}|$  for  $\alpha \in \{0, 1\}$  which returns the number of times  $\alpha$  occurred before  $i$ . On bit vectors, these queries can be answered in constant time [Jac89]. In practice, the currently most space-efficient approach maintaining reasonable query performance is called flat rank [Kur22]. Flat rank only requires 3.51 % additional space. The general idea is to partition the bit vector into blocks and to store the number of 1-bits in the bit vector up to each of the blocks in an array (see [Kur22] and our supplemental for details).

## 2.4. Wavelet Trees

Wavelet trees [GGV03] are compressible index data structures with applications in text indexing [FM00, GNP20], text compression [GVX11, Mak12], and computational geometry [MN06]. They generalize, among others, *access* and *rank* queries from binary alphabets to alphabets of size  $\sigma$  and can answer these queries in  $O(\log \sigma)$  time. We now give a brief overview of wavelet trees along the variants in Figure 2. Due to space constraints, we omit some general details and refer to the excellent surveys on wavelet trees [GVX11, Mak12].

**Structure of a Wavelet Tree.** Each node of the wavelet tree (left in Figure 2) represents a subset of the whole text  $T$  and has up to two children. These subsets contain all characters of  $T$  that are in a subset of the alphabet, i.e. wavelet trees partition the *alphabet*, not the space of the text. The root at the top considers the entire alphabet and thus represents the whole text. Now, the left child of the root represents all characters in the lower half of the alphabet and the right child represents all characters in the upper half of the alphabet. In other words, when the root considers the alphabet  $\Sigma = [1, \sigma]$ , the left child considers  $\Sigma_\ell = [1, \sigma/2]$  and the right child considers  $\Sigma_r = [\sigma/2 + 1, \sigma]$  minus rounding. The remaining nodes are defined recursively and we stop at depth  $\lceil \log \sigma \rceil$ .

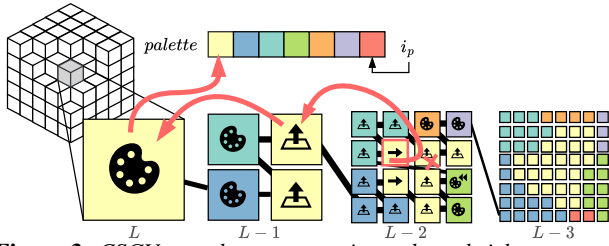
We use bit vectors to represent the characters. If a character is in

the lower half of the alphabet, we mark it with a 0-bit. It is marked with a 1-bit otherwise. At each node, we only look at characters in the subset represented by the node, i.e., the total length of all bit vectors on the same level of the tree is always  $|T|$ . The red path in Figure 2 highlights the query for  $\text{access}^T(8) = T[8]$  starting at position  $i = 8$  in the root node and descending left or right depending on the bit  $\alpha$  at  $i$ . All characters that share the same bit will belong to the same node in the next level. Thus,  $\text{rank}_\alpha(i)$  in the current node determines the next index in the node in the following level. In our example, this descend through the tree iteratively extends the bit-prefix of a given character until all bits are known.  $\text{rank}_\omega^T$  operates similarly: as the same character  $\omega \in \Sigma$  always ends in the same node, a bit vector *rank* in that node is equal to  $\text{rank}_\omega^T$ . To find the node, the direction bit  $\alpha$  in level  $l$  is not obtained through an *access* but is the  $l$ -th bit of the code of  $\omega$  (see our supplemental for details).

In memory, wavelet trees are not stored using pointers: The level-wise wavelet tree concatenates all bit vectors on the same level [CKV24]. A more efficient representation is the wavelet matrix [CNP15], which is a level-wise wavelet tree where the nodes on the same level have been reordered to save one *access* and/or *rank* query per level in practice. Since these are only pointer-less *memory representations* of wavelet trees, we always refer to the structure as *wavelet tree*, as is convention in literature.

**Improving Wavelet Tree Queries.** However, originally designed as an alternative representation of the wavelet tree for large alphabets, the wavelet matrix [CNP15] layout provides also faster queries in practice without any disadvantages. Furthermore, because of their similarity, all the results for wavelet trees are also applicable to wavelet matrices. There has been further effort to improve query performance of wavelet trees (and matrices) using multi-ary trees [FMMN07, CKV24] and wavelet forests [HBG\*24].





**Figure 3:** CSGV encodes segmentation volume bricks as a multi-resolution grid, here in 2D. Each grid node contains an operation determining if it copies the label from a parent  $\triangleleft$ , neighbor  $\rightarrow$ , or the palette  $\oplus$ . CSGV operates along the serialized operation stream (black line). Our random access CSGV-R follows the operations from the accessed node up to a palette operation (red arrows).

**Wavelet Tree Construction.** Babenko et al. [BGKS15] and Munro et al. [MNV16] presented the best sequential construction algorithms that require only  $O(n \log \sigma / \sqrt{\log n})$  time. There exists a lot more (theoretical and practical) work on the efficient construction of wavelet trees in many different models of computation [Shu20, Shu15, FEFS17, dFdS17, CNS11, Tis11, Kan18, EK19, DFK20, DEF\*21, DFKT23]. It should be noted that, to the best of our knowledge, the only GPU implementation of wavelet trees is by NVIDIA as part of their NVBIO library [NPS20].

**Compressing Wavelet Trees.** Wavelet trees can easily be compressed. To this end, we can build the wavelet tree for the Huffman compressed text [Huf52] (right in Figure 2). This type of wavelet tree is called the Huffman-shaped wavelet tree, because the depth of the leaves now depends on the length of the codewords of the characters represented. These Huffman-shaped wavelet trees require only  $|T| \lceil H_0(T) \rceil (1 + o(1))$  bits of space. Here,  $H_0(T)$  denotes the 0-th order entropy, i.e., the least amount of space achievable when encoding each symbol of the text separately. Note that we cannot use Huffman-codes directly, instead we have to use bit-wise negated CHC [DEF\*21], which do not worsen any theoretical guarantees but provide the benefit of smaller codes being lexicographically greater. This allows for us to have trees, where the shorter levels are all on the right-hand side of the tree, simplifying navigating the tree. For our special case in Figure 2, no right children exist at all because of the given CHC pattern. Most *access* queries terminate in early depths. Section 3 includes pseudo code for our streamlined *access* and *rank* queries. In this work, we use wavelet trees to compress our segmentation volume encoding in form of such streamlined Huffman-shaped wavelet matrices, supporting random access and *rank* queries.

## 2.5. Fast Compressed Segmentation Volumes

Segmentation volumes are 3D grids storing integer labels for each voxel. Voxels with the same label usually form a contiguous region and partition the space into separate object regions. While segmentation volumes may assign multiple labels to a single voxel, we focus on the most common case where each voxel  $\mathbf{v} \in \mathbb{N}^3$  is assigned to exactly one label  $\lambda \in [0, 2^{32})$ . CSGV [PD24] is a brick-wise compression for such volumes: Therefore, the volume is split into equally sized bricks of  $b^3$  voxels with  $b = 2^i$ ,  $i \in \mathbb{N}$ ;  $b$  typically being 16, 32, or 64. Each brick is (de-)compressed completely independently. For

each brick, CSGV first builds a multi-resolution grid of its volume by halving  $b$   $i$ -times; the resulting elements are denoted as nodes (Figure 3). This grid represents  $\lfloor L = \log_2(b) \rfloor + 1$  levels-of-detail (LODs) indexed with  $l \in [0, L]$ . Each node is assigned the most frequent label of its  $2^3$  child nodes on the next finer LOD which cover the same region in the segmentation volume. The grid nodes are then serialized from coarsest to finest LOD and along a Morton Z-order curve within an LOD.

CSGV encodes the grid nodes' labels along this indexing order with one operation per node, and stores a palette of labels. Instead of storing the nodes' indices into the palette explicitly, an index pointer  $i_p$  is initialized to the first palette entry and incremented each time the next palette entry is read. The possible operations for a node are

- $\triangleleft$  copy the label of the parent node,
- $\rightarrow$  copy the label of an axis-aligned neighbor node,
- $\oplus$  read the label at  $i_p$  from the palette and advance  $i_p$ ,
- $\oplus^{\delta}$  re-read the label at  $i_p - 1$  from the palette,
- $\oplus^{\delta}$  re-read the label at  $i_p - \delta$  from the palette.

The neighbor operations  $\rightarrow = \{\rightarrow, \uparrow, \nearrow\}$  always refer to the neighbor nodes outside of the current  $2^3$  voxel block, i.e. neighbors with a different parent. If the labels of these neighbors have not yet been decoded, their parents are referenced instead (red cross in Figure 3).  $\oplus^{\delta}$  requires additionally storing the value  $\delta$  in the operation stream.

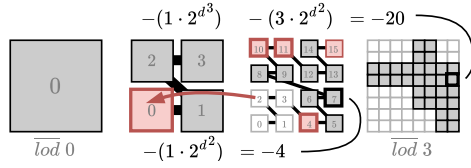
In a plain CSGV Nibble-wise encoding, 3 bits per node suffice to store its operation. An additional *stop bit* is used to mark nodes for which all child nodes have the same label. These child nodes can be omitted in the encoding stream resulting in an Octree-like sparsification. For better compression, CSGV uses a variable bit-length encoding (range ANS encoder [Dud13]) to compress the brick's operation stream. Note that CSGV always decodes a brick fully up to a certain LOD: The brick's operation stream is traversed from the coarsest LOD up to the last node of the target LOD. During processing of the node operations, the output array temporarily stores the labels set by coarser LODs to keep track of parent labels and constant regions from stop bits.

## 3. Random Access Segmentation Volume Compression

The ANS stream compression used in CSGV makes random access impossible. Therefore, we will compact the stream of node operations  $T[i] : \mathbb{N} \mapsto \Sigma$  with a Huffman-shaped wavelet tree, which enables variable bit-length compression and random access. The operation  $\oplus^{\delta}$  in CSGV requires storing the  $\delta$  values as additional values in the operation stream. As these  $\delta$ -values increase the number of different occurring values in the stream, we omit this operation in our CSGV-R and only support the most frequent back reference distance  $\delta = 1$  that is directly encoded as  $\oplus^1$ . That is, our alphabet is  $\Sigma = \{\triangleleft, \rightarrow, \uparrow, \nearrow, \oplus, \oplus^1\}$  with a size of  $|\Sigma| = 6$ .

Variable bit-length compression is beneficial as the operations used in CSGV occur in vastly different frequencies. We use Huffman-shaped wavelet trees mapping symbols to bit patterns, their bit-wise negated canonical Huffman codes (CHC) [DEF\*21]. The CHCs are constructed from the Huffman tree, i.e. from the operation frequencies. In CSGV, parent references are the most frequent symbols, neighbor references are less frequent, and palette accesses and back





**Figure 4:** Stop bits (red nodes) can create a negative offset within the operation stream for a node. Stop bits on coarser levels that are in an earlier region along the Z-curve remove  $(2^d)^{\Delta_L}$  node index predecessors. Nodes that are in a removed region must use the index of the parent that sets the stop bit instead (red arrow). Numbers inside nodes are their Morton codes  $n$  inside their  $\overline{LOD}$ .

**Algorithm 1**  $idx(\&n, \&\bar{l})$  computes the index of a multi-grid node given by its Morton code  $n$  in its  $\overline{LOD}$   $\bar{l}$  within the operation stream.  $n$  and  $\bar{l}$  are passed by reference.  $T_i$  is the stream index of the first node in each  $\overline{LOD}$ ,  $S$  is the stop bit vector.

**Input** grid node given as the Morton code  $n$  in its  $\overline{LOD}$   $\bar{l}$

**Output** operation stream index of the node

```

1: offset  $\leftarrow 0$ 
2: for  $\bar{l}' = 0 \dots \bar{l}$  do
3:    $n_{parent} \leftarrow \lfloor n/2^{3(\bar{l}-\bar{l}')} \rfloor$   $\triangleright$  parent node index in its  $\overline{LOD}$ 
4:   if  $access^S(T_{\bar{l}'} + n_{parent} - offset) = 1$  then
5:      $\bar{l} \leftarrow \bar{l}'$ 
6:      $n \leftarrow n_{parent}$   $\triangleright$  Return parent if it sets a stop bit
7:   break
8:   offset  $\leftarrow offset + rank_1^S(T_{\bar{l}'} + n_{parent} - offset)$ 
9:   offset  $\leftarrow offset - rank_1^S(T_{\bar{l}'})$ 
10: return  $(T_{\bar{l}} + n - offset)$ 

```

references occur the least. The pattern is consistent across data sets and domains [PD24], which makes it possible to determine Huffman codes once. Conveniently for Huffman coding, the operation distribution roughly follows  $(1/2)^{1\dots 7}$ . Figure 2 (right) shows the resulting CHCs for our alphabet  $\Sigma$ . The maximum length of a bit pattern is 5 which means that our Huffman-shaped wavelet tree uses 5 levels. Note that the CHCs follow a simple layout, similar to RICE coding [RP71] where the number of consecutive 0 bits determines the operation. For this reason, our wavelet trees contain no right child nodes, i.e. any 1 bit terminates a symbol (Figure 2 middle).

**Wavelet Tree Compression.** When compressing a volume brick, we first encode its label multi-grid from coarsest to finest level into a list of operations as in CSGV. Recall that we do not use  $\otimes$ , and replace it by  $\oplus$  instead, possibly creating additional palette entries. The operation stream  $T$  is subsequently compressed as a Huffman-shaped wavelet tree with our hard coded CHCs. We store it as a wavelet matrix [CNP15] and refer to its resulting level-wise concatenated bit vector as  $W$  and to the level start indices inside  $W$  as  $W_0 \dots W_4$ .  $W$  is constructed with the prefix counting algorithm [DEF\*21]. Next to  $W$  we store a flat rank structure [Kur22] for constant time  $rank$  queries.

**Stop Bits.** CSGV introduces two variants of each operation where a marker stop bit determines if the covered region by a grid node is constant in all finer levels. Respective nodes in the finer levels are

then excluded in the encoding, resulting in storage akin to Octrees. We decouple these stop bits from the operation codes to keep the alphabet size and depth of the wavelet trees limited. The stop bits  $S[i] : i \mapsto \{0, 1\}$  are stored in a separate bit vector that follows after the operation list with its own flat rank data structure. We implement two variants of CSGV-R, with and without using  $S$ .

**Brick Headers.** The brick headers store the start offset of each CSGV LOD within the operation and stop bit lists as  $T_0 \dots T_L$ , the palette size, and the information to query the wavelet matrix. The latter includes the number of 1 bits before each level in  $W$  as  $Z_1[0] \dots Z_1[4]$ . Figure 5 shows the final memory layout of a compressed brick not true to scale. Note that the levels of the wavelet tree  $I$  are *not* the LODs of the multi-resolution grid  $I$  in the CSGV encoding. Queries for all CSGV LODs start within the first wavelet tree level.

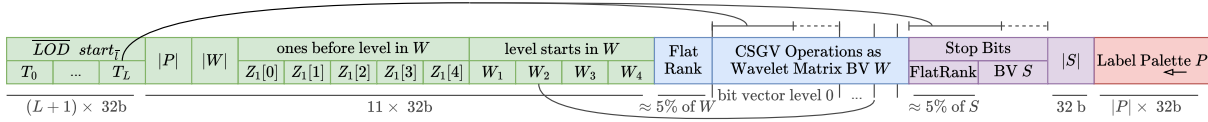
### 3.1. Random Access Decoding

Next we discuss the fundamental differences of the non-random access CSGV and our random access CSGV-R. CSGV operates serially within a brick: To access the label of a brick node  $T[i]$ , all previous nodes  $T[j], j < i$  must be decoded first. In practice, full bricks are decoded up to a certain LOD and cached, and if all of the brick's nodes will be accessed later, costs amortize quickly. However, if only few nodes will be accessed, for example due to occlusion in a renderer or when a task requires sampling random voxels globally, the brick-wise caching poses a significant memory overhead.

Mainly two reasons make the serial decoding necessary for CSGV: The variable bit-length ANS compression makes a direct mapping of an operation stream index  $i$  to its memory location impossible. And no explicit palette indices are stored for  $\otimes$  and instead the index is effectively the number of preceding  $\otimes$  in the operation stream. Obtaining the  $i$ -th operation in  $T$  is an  $access(i) = T[i]$  query. Computing the palette index at operation  $i$  where  $T[i] = \otimes$  translates to  $rank_{\otimes}(i) = |\{j < i : T[j] = \otimes\}|$ . Our wavelet tree answers both queries in constant time given our constant  $\Sigma$ .

To resolve parent and neighbor references, CSGV uses the temporary information stored in the partially decoded brick. Random accesses are not possible as such a partial decoding does not exist in this case. In CSGV-R we instead start at the accessed grid node and follow the indirections introduced by  $\triangleleft$  and  $\rightarrow$  as a chain of operations from grid node to grid node until either  $\otimes$  or  $\oplus$  is reached (red arrows in Figure 3). Then, a  $rank_{\otimes}$  query returns the index to the brick palette at which the label of the accessed node is looked up.

Grid nodes can either be referred to by their index in the operation stream  $i$ , or by their coordinate in the grid. In the latter case, the coordinate is a tuple of the node's 1D Morton code  $n$  of its coordinate within its  $\overline{LOD}$   $\bar{l}$ . The  $\overline{LOD}$  0 is the coarsest level consisting of one node and  $L$  is the finest  $\overline{LOD}$  with  $b^3$  nodes. Figure 4 shows  $n$  within the nodes. Note that the Morton code of the parent of node  $n$  is  $\lfloor n/2^d \rfloor$  where  $d = 2$  in 2D. The operation of a node is obtained as  $access(i) \iff access(idx(n, \bar{l}))$ .  $idx$  returns the operation stream index  $i$  for the tuple  $(n, \bar{l})$ .  $idx$  computes the negative offset from  $n$  to  $i$  within a level that occurs because earlier nodes were removed by stop bits on coarser levels (Figure 4). Recall that the stop bit



**Figure 5:** Memory layout of a compressed brick (not true to scale). The LOD operation starts are the indices  $i$  at which new CSGV multi-grid levels start in  $T$  (not to be mistaken for the wavelet matrix level start indices  $W_i$ ). The wavelet matrix bit vector  $W$  and stop bit vector  $S$  are accompanied by a flat rank structure for constant time rank queries. The palette of uncompressed labels is stored in reverse order as in CSGV.

#### Algorithm 2 Operation Stream Wavelet Matrix *access*

**Input** operation stream index  $i$ , wavelet matrix bit vector  $W$   
from brick header:  $W_1$  level starts,  $Z_1[1] = \text{rank}_1^W(W_1)$

**Output** operation  $\omega = T[i] \in \{\triangle, \rightarrow, \uparrow, \searrow, \oplus, \otimes\}$

```

1: for  $l = 0 \dots 4$  do
2:    $\text{bit} \leftarrow \text{access}^W(W_1 + i)$  ▷ bit vector access
3:   if  $\text{bit} = 1$  then ▷ any 1 bit terminates symbol
4:     return  $l$  ▷ level = |CHC 0 bits| =  $\omega$ 
5:    $\text{ones}_{\text{before}} \leftarrow \text{rank}_1^W(W_1 + i) - Z_1[1]$ 
6:    $i \leftarrow i - \text{ones}_{\text{before}}$ 
7: return 5 ▷ five consecutive 0 bits are  $\oplus$ 

```

vector  $S[i]$  stores for each node  $i$  if its covered region is missing in the operation stream on all finer levels.  $\text{rank}_1^S(i)$  returns the number of 1 bits, i.e., stop bits, before  $i$  in  $S$ . The brick headers store the stream index of the first grid node,  $n = 0$ , in each LOD, denoted  $T_0 \dots T_L$ . Here, the mapping of a grid node to its index is

$$i = \text{idx}(n, \bar{l}) = T_{\bar{l}} + n - \sum_{l < \bar{l}} \text{rank}_1^S(P_l(n, \bar{l})) \cdot 2^{3(\bar{l}-l)}$$

where  $P_l(n, \bar{l})$  is the index of the (grand-)parent node of  $n$  in LOD  $\bar{l}$ . While it initially may seem that  $P_l$  introduces a recursion by requiring an operation index mapping itself, it can be computed iteratively along with the inner sum in  $\text{idx}(n, \bar{l})$  as shown in Algorithm 1. Because all inner operations require constant time,  $\text{idx}(n, \bar{l})$  can be answered in  $O(L) = O(\log b)$  time.

### 3.2. Wavelet Tree Implementation

On the GPU, our CSGV-R are buffers of concatenated compressed bricks, each with the layout from Figure 5. Vulkan buffer device addresses allow to pass references to brick bit vectors and headers to the decoding functions. Our bit vectors and flat ranks are stored as arrays of 64 bit words where the bit index order inside a word goes from the least to the most significant bit. Our wavelet trees are implemented as Huffman-shaped wavelet matrices as in Figure 2.

**Wavelet Matrix Access.**  $\text{access}^W(i)$  of the  $i$ -th bit in a bit vector  $W$  is computed as  $(\text{words}[i / 64] \gg (i \& 63)) \& 1$  and uses one integer division, one bit shift and two bit-wise ands. Our RICE-coding like CHC (Section 3) results in the operation number simply being the number of zeros before the first 1 bit in a Huffman-shaped wavelet tree symbol. Additionally, this completely eliminates the 1 branch and the tracking of interval starts from the general Wavelet Matrix *access* query [CNP15] as laid out in Algorithm 2.

#### Algorithm 3 Operation Stream Wavelet Matrix *rank*

**Input** operation stream index  $i$ , wavelet matrix bit vector  $W$   
from brick header:  $W_1$  level starts,  $Z_1[1] = \text{rank}_1^W(W_1)$

**Output** number of  $\oplus$  in  $T$  before  $i$ :  $\text{rank}_{\oplus}(i)$

```

1: for  $l = 0 \dots 4$  do
2:    $\text{ones}_{\text{before}} \leftarrow \text{rank}_1^W(W_1 + i) - Z_1[1]$ 
3:    $i \leftarrow i - \text{ones}_{\text{before}}$ 
4: return  $\text{ones}_{\text{before}}$ 

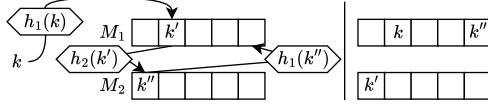
```

**Wavelet Matrix Rank.** We use an adapted flat rank [Kur22] for constant time *rank* queries on our bit vectors at 5% space overhead (details in the supplemental). Wavelet matrix *rank* only occurs for  $\oplus$ . Our *rank* implementation collapses from the generalized form (cf. [CNP15]) to the efficient implementation shown in Algorithm 3: Hardcoding the character's CHC (00001) removes all branching. A generalized wavelet matrix implementation must track where tree node intervals start within a level [CNP15]. We can optimize out all of this: a single interval exists per level ( $0^l$  prefix) which starts at the level start index  $W_1$  in  $W$ . We unroll the loop in our implementation.

### 4. Rendering of CSGV-R Encoded Volumes

We implemented a segmentation volume renderer using ray marching in Vulkan compute shaders. One thread per pixel marches along a ray from the camera through the volume using digital differential analyzer (DDA) traversal [AW87]. We access brick LODs so that one decoded voxel approximately maps to one pixel in the output image. For coarser LODs, the DDA step size is increased accordingly. The label of an accessed brick is mapped to opacity and color by a transfer function. Opacity values above a threshold are interpreted as surface interactions where a local diffuse shading model is evaluated and the thread either terminates or continues to pursue an indirect shadow or ambient occlusion ray, depending on the shading mode.

For accessing voxel labels, CSGV caches visible bricks, fully decoded up to an LOD. When their renderer requests a voxel from a brick that is not available, the brick is scheduled for decoding in the next frame; akin to shading atlas streaming [MVD\*18] bricks are assigned to free regions in the cache. Our CSGV-R is not dependent on a cache and the renderer can directly access voxels from encoded bricks. However, as is discussed in section 5.2, operating the rendering completely in the compression domain is slow and should thus only be used if memory budgets are strict. For other cases, we introduce a voxel cache with variable size to store decoded labels of single voxels. It operates without the one frame latency of CSGV as voxels not in the cache can be decoded directly.



**Figure 6:** Cuckoo hashing uses two hash functions  $h_1(k)$  and  $h_2(k)$  mapping keys into distinct positions in two hash tables  $M_1$  and  $M_2$ . When keys are inserted in one table, starting from  $M_1$ , ejected elements are recursively swapped between their two positions.

#### 4.1. Voxel Caching

For our optional voxel cache we use a hash map which stores labels  $\lambda$  for tuples of voxel coordinates  $\mathbf{v} \in \mathbb{N}^3$  (measured in the finest resolution) and requested LODs  $l$ . We use Cuckoo hashing [PR04] to map the tuple  $(\mathbf{v}, l)$  to positions in the hash map  $p$ . Cuckoo hashing defines two hash functions for each element,  $h_1(k)$  and  $h_2(k)$  mapping element keys  $k$  to positions in two different hash tables as  $M_1[h_1(k)]$  and  $M_2[h_2(k)]$ . As any element can only be found at either of those two positions, the lookup function operates in constant time. When inserting an element, it is stored at  $M_1[h_1(k)]$ . If this position was already occupied by an element  $k'$ ,  $k'$  is moved to its second location  $M_2[h_2(k')]$ , possibly ejecting another element that is then moved from  $M_2$  to its other position in  $M_1$  and so on (Figure 6). This ejection loop continues either until an element is inserted into an unoccupied position, a predetermined loop depth is reached, or a cycle is detected (an element is inserted into a position that it occupied before). To construct the hash functions, Pagh and Rodler [PR04] use an XOR combination of three functions from the (c,k) universal family [CW77] given by Dietzfelbinger et al. [DHKP97]:

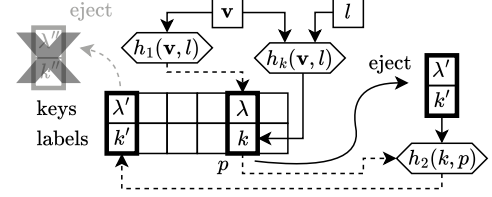
$$h^a(x) = (ax \bmod 2^{32}) / 2^{32-m}$$

where  $a$  is an odd number and  $m$  is the most significant bit (MSB) of the power-of-two hash table size. As our query tuples are three or four-dimensional, we opt for the pcg family [O'N14, JO20] instead. In the original Cuckoo hashing [PR04], the keys in the table must be identical to the access tuples. For our tuple  $(\mathbf{v}, l)$  this would require four 32 bit integers. Pagh and Rodler [PR04] propose to store such long keys outside the table and smaller pointers to the keys inside the table. In the following, we detail our specific approach which allows us to use a small hash of the tuple as the key instead.

**CSGV-R Cuckoo Caching.** We use one single table  $M$  to hold  $M_1$  and  $M_2$  at the same time, as discussed by Pagh and Rodler [PR04].  $M$  is an array storing the label  $\lambda$  and a 32 bit key  $k$  to identify hash collisions at each position. The first hash function for our method is used to determine the first position of a 4D tuple  $(\mathbf{v}, l)$  in  $M$  as

$$h_1(\mathbf{v}, l) = h^{19}(\mathbf{s}.x) \oplus h^{47}(\mathbf{s}.y) \oplus h^{101}(\mathbf{s}.z) \bmod |M|$$

where we compute  $\mathbf{s} = \text{pcg3d}(\lfloor v/2^l \rfloor \cdot 2^l)$  using the pcg3d 3D hash function [JO20] (see Figure 7) and  $\oplus$  is a bitwise XOR. Rounding voxel coordinates down to powers-of-two forces hash table collisions for voxels from different LODs belonging to the same spatial region. This is desirable as we will never access different LODs of a region at the same time during rendering. Shader vector arithmetic allows to compute up to four  $h^a(x)$  simultaneously. At  $M[h_1(\mathbf{v}, l)]$ , we store the voxel's label  $\lambda$  and the 32 bit key  $k$  of the 4D lookup position



**Figure 7:** Insertion of a label  $\lambda$  for voxel  $\mathbf{v}$  in LOD  $l$  with our adapted Cuckoo hashing.  $(\lambda', k')$  is ejected from its first position and can be swapped into its second positions.  $(\lambda'', k'')$  cannot be moved to its alternative location as it is ejected from its second position.

$(\mathbf{v}, l)$ . The key is computed with another hash function as

$$k = h_k(\mathbf{v}, l) = h^5(\mathbf{t}.x) \oplus h^{149}(\mathbf{t}.y) \oplus h^{61}(\mathbf{t}.z) \oplus h^{2887}(\mathbf{t}.w)$$

with  $\mathbf{t} = (\text{pcg4d}(\lfloor v/2^l \rfloor \cdot 2^l), l)$  being a 4D vector. We use the MSB of  $k$  to track if an element is stored at its first (set to 0) or second position (set to 1). The alternative hash table position of an element is determined with  $\mathbf{u} = \text{pcg2d}(h_k(\mathbf{v}, l), h_1(\mathbf{v}, l))$  as

$$\tilde{h}_2(\mathbf{v}, l) = h^7(\mathbf{u}.x) \oplus h^{31}(\mathbf{u}.y) \oplus h^{97}(\mathbf{u}.x) \oplus h^{173}(\mathbf{u}.y) \bmod |M|.$$

When an element  $(\lambda, k)$  inserted at  $p$  ejects a previous element  $(\lambda', k')$  for which  $p$  is the first position, the ejected element is moved to its second position  $p'$ . Given our definition of  $\tilde{h}_2$  we can compute  $p' = h_2(k', p)$  with  $\mathbf{u} = \text{pcg2d}(k', p)$ , set the MSB of  $k'$  to 1 and insert  $(\lambda', k')$  at  $p'$ . If  $p$  already is the second position for  $(\lambda', k')$ , then the element cannot be moved back to its first position since the input parameters for  $h_1$  are not available. Instead, the element is removed until the renderer accesses the associated voxel for the element again. Then a cache miss will occur and  $(\lambda', k')$  will be re-inserted. Note that elements still follow the Cuckoo hashing pattern of alternating between their first and second positions, and lookups operate in constant time accessing at most both positions. We limit the ejection recursion to a depth of 1 to terminate the Cuckoo hashing ejection loops earlier and thus limiting thread divergence.

#### 4.2. Empty Space Caching

A naive implementation of the above described hashing will suffer from the fact that cache space is quickly exceeded if many voxels are mapped to transparent by the transfer function. As the ray marcher cannot determine if a label is visible before reading it from the cache or decoding it, an empty space skipping solution is required to reduce the number of accessed voxels. As in CSGV we can easily skip bricks which do not contain any visible labels by testing and marking the label palettes in another shader before rendering [PD24]. However, this is not yet sufficient and we thus add a more fine-granular empty space skipping bit vector  $E$  that stores one bit for  $2^3$  voxels, which we refer to as *empty block*, on the finest LOD of the full volume. Each bit in  $E$  is 1 if and only if all voxels in that empty block have an invisible label. For such invisible empty blocks,  $E$  stores only  $\frac{1}{8}$  bits per voxel, while the hash table  $M$  would require 64 bit per invisible voxel (32 for  $\lambda$  and 32 for  $k$ ). Note that for partially visible empty blocks, all of its voxels have to be decoded and cached in  $M$ .

We construct  $E$  on-the-fly during rendering: It is initialized to  $E = (000 \dots 000)_2$  and reset whenever the visibility transfer function



**Algorithm 4** getLabelOfVoxel(uvec3  $\mathbf{v}$ , uint  $l$ ). We use a Cuckoo hashing variant for voxel label caching and an optional empty space skipping bit vector (gray). Insertions in  $E$  and  $M$  are atomic.

---

**Input** voxel coordinate in finest LOD  $\mathbf{v}$ , accessed LOD  $l$   
**Output** label  $\lambda$

```

1: if brickInvisible( $\mathbf{v}/b$ ) then           ▷ brick skipping as in CSGV
2:   return INVISIBLE                     ▷  $b$  is the brick size
3:  $e \leftarrow$  index of  $\mathbf{v}$ 's empty block in  $E$ 
4: if  $\text{access}^E(e) = 1$  then               ▷ Check if  $E$  marks  $\mathbf{v}$  invisible
5:   return INVISIBLE
6:  $p_1 \leftarrow h_1(\mathbf{v}, l)$                ▷  $\leq 2$  lookups for  $(\mathbf{v}, l)$  in  $M$ 
7:  $k \leftarrow h_k(\mathbf{v}, l) \& (0111 \dots 1111)_2$ 
8:  $(k_1, \lambda_1) \leftarrow M[p_1]$ 
9: if  $k_1 = k$  then
10:  return  $\lambda_1$ 
11: else
12:   $p_2 \leftarrow h_2(k, p_1)$ 
13:   $(k_2, \lambda_2) \leftarrow M[p_2]$ 
14:  if  $k_2 = (k \mid (1000 \dots 0000)_2)$  then
15:    return  $\lambda_2$ 
16:  $\lambda \leftarrow \text{decodeLabel}(\mathbf{v}, l)$      ▷ CSGV-R random access
17: if  $\neg \text{labelVisible}(\lambda)$  then         ▷ Check if voxel set fully invisible
18:    $\text{allInvisible} \leftarrow \text{True}$ 
19:   for all  $\mathbf{v}'_0$  with empty block index  $e$  do
20:     if  $\text{labelVisible}(\text{decodeLabel}(\mathbf{v}'_0, 0))$  then
21:        $\text{allInvisible} \leftarrow \text{False}$ 
22:   if  $\text{allInvisible}$  then
23:      $E[e] \leftarrow 1$ 
24:   return INVISIBLE
25:  $M[p_1] \leftarrow (k, \lambda)$            ▷ Insert in  $M$ , possibly swap ejected  $(k_1, \lambda_1)$ 
26: if  $(k_1, \lambda_1) \neq \emptyset \wedge (k_1 \& (1000 \dots 0000)_2) = 0$  then
27:   $p' \leftarrow h_2(k_1, p_1)$ 
28:   $M[p'] \leftarrow (k_1 \mid (1000 \dots 0000)_2, \lambda_1)$ 
29: return  $\lambda$ 

```

---

changes. Algorithm 4 contains our voxel access method, including the handling of the optional empty space bit vector  $E$  in gray. Putting everything together: we decode a voxel  $(\mathbf{v}, l)$  if  $\mathbf{v}$  is not inside a completely invisible brick not yet marked as empty in  $E$ , and not stored in the label hash map  $M$ . When such a newly decoded voxel is not visible, all voxels that share its empty block are decoded as well. This is done on the finest LOD regardless of  $l$ . If none of them is visible, the empty block is marked invisible in  $E$  for later accesses. Otherwise,  $M$  is used to cache the label of  $(\mathbf{v}, l)$  as before.

## 5. Evaluation

We evaluate the implementation of our method on a system using an AMD Ryzen 7 5800x 8-core CPU with 64GB RAM and an RTX 4070 TI Super GPU with 16GB VRAM at fixed GPU clock speeds. We will release the source code of our CSGV-R compression. The evaluation data sets are shown in Figure 1 and listed in Table 1:

- CELLS [RBS20] is a densely labeled Cellular Potts Model [GG92]

Data Set	Voxels	Labels	Orig. Size [GB]
CELLS	$1000 \times 1000 \times 1000$	1,067,196	4.000 (4B)
FIBER	$1579 \times 1092 \times 1651$	26,372	5.694 (2B)
H01	$5120 \times 6144 \times 6144$	1,296,889	773.094 (4B)
AZBA	$470 \times 1224 \times 670$	204	0.385 (1B)

**Table 1:** Dimensions and sizes of the evaluated segmentation volumes. Bytes per voxel in the uncompressed volume given in ( ).

- tumor growth simulation. We only show tumor and blood cells, but label regions of roughly  $10^3$  voxels occupy the volume fully.
- FIBER [MSJ\*22] is an X-ray scan of a fiber reinforced polymer with instance segmented glass fibers. Elongated regions span multiple bricks and create complex partial visibility during rendering.
- H01 is a large subset of the H01 petavoxel fragment of a human cerebral cortex [SCJB\*24]. It is a typical example of connectomics volumes with larger but complex and interleaving labeled structures, and vast volume dimensions.
- AZBA [KSY\*21] is a fully segmented zebrafish brain with characteristics commonly seen in medical data sets: a smaller number of labeled regions with a high variation in size. We render it with large semi-transparent areas creating high voxel access loads.

### 5.1. Compression Rates

Table 2 shows results for our CSGV-R compression in different configurations and a comparison with CSGV. Larger  $b$  improve compression rates for both, which is due to the more expensive encoding at brick borders and label duplicates in palettes. When not using any variable bit-length encoding, the operation stream encodes the volumes for a brick size  $b = 64$  with compression rates between 6.4% and 1% (CSGV Nibble). With the non-random access ANS variable bit length coding, improved rates are between 2.8% and 0.4%, and close to the entropy (CSGV rANS). Our random access wavelet matrix compression (CSGV-R+stop bits) yields compression between 4.4% and 0.6%. ANS achieves better compression as its fractional bit encoding can follow relative symbol frequencies more closely. Our Huffman codes encode operations only with an integer number of bits. Furthermore, CSGV encodes the combination of stop bit and operation code in a single combined symbol while we store the stop bits in a separate bit vector to limit wavelet tree depth, and thus query times. Encoding the operation stream as a wavelet matrix is computationally more expensive than with ANS and results in longer compression times. With stop bits, fewer brick nodes exist and must be compressed in total which improves timings.

Comparisons with other segmentation volume compressors are shown in (Table 3). General purpose compressors like gzip used in the hdf5-format [HDF] achieve sub-par compression results. Compresso achieves strong compression when paired with a global LZMA but then random access is not possible. CSGV [PD24] achieves strong compression and at least supports random access to bricks, however, random voxel access or compression domain queries are not supported. Due to its simple brick-wise paletting, Neuroglancer [Goo16] is a fast format with random access down to the voxel level; however, its compression rates are the worst of all evaluated methods. Our CSGV-R with the stop bit vector achieves compression rates that are roughly twice of the CSGV rates, but still significantly better than Neuroglancer, the only other true random ac-

	b	CSGV Nibble			CSGV rANS			CSGV-R			CSGV-R + stop bits		
		CR	Time (s)	GB/s	CR	Time (s)	GB/s	CR	Time (s)	GB/s	CR	Time (s)	GB/s
CELLS	16	6.833%	2.376	1.684	3.436%	2.016	1.984	6.111%	3.47	1.153	5.528%	2.549	1.569
	32	6.561%	<b>1.773</b>	2.256	2.996%	1.781	2.246	5.489%	3.211	1.246	4.653%	2.308	1.733
	64	6.425%	2.473	1.618	<b>2.802%</b>	2.412	1.658	5.257%	3.858	1.037	4.411%	2.919	1.37
FIBER	16	2.97%	3.401	1.674	1.488%	3.95	1.442	9.031%	8.514	0.669	2.899%	4.018	1.417
	32	2.558%	<b>2.464</b>	2.311	0.979%	2.585	2.202	8.299%	7.539	0.755	1.759%	2.896	1.966
	64	2.496%	3.883	1.466	<b>0.894%</b>	3.978	1.431	8.379%	9.299	0.612	1.593%	4.338	1.313
H01	16	2.83%	276.978	2.791	1.616%	283.78	2.724	5.005%	594.828	1.3	2.644%	348.655	2.217
	32	2.608%	<b>228.101</b>	3.389	1.361%	231.258	3.343	4.449%	539.476	1.433	2.005%	279.779	2.763
	64	2.566%	312.467	2.474	<b>1.31%</b>	307.957	2.51	4.336%	619.911	1.247	1.882%	345.899	2.235
AZBA	16	1.871%	0.349	1.103	1.399%	0.703	0.548	17.656%	1.124	0.343	3.314%	0.376	1.024
	32	1.079%	<b>0.279</b>	1.383	0.495%	7.532	0.051	16.061%	1.023	0.377	0.958%	0.292	1.319
	64	0.968%	0.47	0.821	<b>0.366%</b>	0.984	0.392	18.1%	1.431	0.269	0.648%	0.469	0.822

**Table 2:** Compression rates of different data sets and brick sizes  $b$  and compression times when using 16 threads in parallel. CSGV Nibble uses 4 plain bit per node, rANS uses range ANS coding with two frequency tables as in [PD24]. We evaluate our CSGV-R with and without using the stop bit vector while others always include stop bits. Compression rates are given as compressed size / original size fraction.

cess format. In some data sets, e.g. AZBA and FIBER, our CSGV-R even outperforms Compresso with LZMA.

## 5.2. Rendering

We evaluate our random access CSGV-R in a DDA-ray marching renderer with a wide range of configurations. In Table 4 We list average render times per frame measured for a camera path around the data sets and compare it to CSGV. We use  $b = 32$  except for H01 where  $b = 64$ . When using a cache during rendering, the cache size is 1 GiB except when rendering H01 with CSGV where a larger cache is required for which we chose 4 GiB. For shadow and ambient occlusion rays, we cast one indirect ray per pixel per frame. Rendering with CSGV-R generally outperforms the brick-serial decoding of CSGV, inter alia by not requiring separate caching [MVD\*18] and decoding shader stages (*no cache* or *voxel cache*) which also reduces implementation complexity. Further, it reduced the number of unnecessarily decoded voxels and achieves a higher thread coherence during decoding. While using stop bits improves compression ratios, the additional querying of the stop bit vector for the operation stream index computation puts a strain on the decoding performance. Especially for data sets like CELLS, in which label regions are small and few stop bits are set, frame times are almost doubled. The different caching and decoding modes are discussed in the following.

**Random Access Decompression.** As our CSGV-R compression offers true random access, we can perform rendering without any cache (*no cache*). However, such voxel accesses come at a cost: chasing along the operations up to a palette operation, traversing the wavelet matrix for each of these *access* as well as the final palette index *rank* query. As a consequence, caching decoded individual voxels (*voxel cache*) with our voxel hash table and custom Cuckoo hashing (Section 4.1) significantly improves performance, in some cases up a factor of 5. As a large portion of the voxel cache cells contain invisible labels, we introduced the optional empty space bit vector that we fill on-the-fly during rendering (*voxel cache (es)*). While this improves render times for CELLS and AZBA, we measured minimal changes for FIBER, and performance even decreases significantly for

H01. This is presumably due to the large dimensions of H01 which results in many distant memory reads in the longer empty space bit vector. Moreover, setting the empty space bit flags requires querying  $2^3$  voxels in the finest and most expensive LOD, while common accesses in H01 are usually to coarser and faster accessible LODs.

**Brick-wise Decompression Stage.** CSGV only supports caching full bricks [PD24], decompressed by a separate shader up to a requested LOD with a one frame latency (*brick cache*). Cache regions for the bricks in their requested LODs are assigned as in shading atlas streaming [MVD\*18]. The decoding shader must decompress bricks serially using one thread per brick, i.e. each thread in a GPU wavefront accesses different global memory regions and different work loads leading to thread divergence. We implemented an alternative brick-wise decompression shader stage for our CSGV-R where one wavefront cooperatively decompresses one brick for CSGV-R modes): Threads in the wavefront iterate over output voxels in the requested LOD and read its volume label with random access as usual. To further improve performance, the brick's wavelet tree is copied to shared memory before decoding (*brick cache<sup>c</sup> (sm)*). For FIBER, this cooperative decoding of bricks achieves the fastest rendering.

## 5.3. CSGV Caching Overhead and Latency

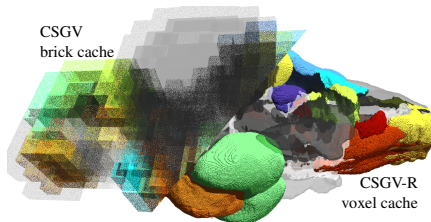
While CSGV achieves smaller compressed volumes, it always requires extra memory during rendering for the cache of visible bricks. CSGV-R encoded volumes do not have this overhead and can be rendered as long as they fit into VRAM. The first time a brick in CSGV is accessed, it is not present in the cache. Before it gets decompressed, a random sample from the brick palette or the most frequent label in the brick (the first palette entry) can be used as a placeholder. This, however, results in visible popping artifacts (Figure 8). In the worst case, the cache is not large enough to store all accessed bricks when rendering a frame. In fact, the CSGV cache barely fits all bricks for ambient occlusion configurations. Deeper traversal as in path tracing will not allow decoding all visible voxels in CSGV. In any case, especially when casting indirect rays for ambient occlusion, the latency of at least one frame reduces the number

data set	size/#labels	hdf5	Compresso	+ LZMA	Neuroglancer	CSGV rANS	CSGV-R	+stop bits
CELLS	4.0GB	7.221%	8.337%	<u>2.753%</u>	13.622%	2.802%	5.257%	<b>4.411%</b>
	1M labels	21.8s	33.0s	138.5s	10.8s	15.9s	25.5s	17.7s
FIBER	5.7GB	3.051%	26.70%	5.861%	3.658%	<u>0.892%</u>	8.379%	<b>1.593%</b>
	26K labels	47.4s	139.2s	654.1s	11.0s	19.4s	61.8s	18.4s
H01*	4.295GB	3.678%	2.284%	<u>0.342%</u>	3.882%	1.273%	4.313%	<b>1.830%</b>
	5.7K labels	18.0s	19.6s	53.4s	6.3s	9.0s	23.0s	9.2s
AZBA	0.385GB	2.166%	88.357%	1.901%	2.801%	<u>0.366%</u>	18.100%	<b>0.648%</b>
	204 labels	7.0s	9.0s	47.2s	1.2s	2.3s	9.1s	1.9s

**Table 3:** Comparison of compression rates and single-threaded compression time (excluding file import) of our CSGV-R and other compressors. General purpose compressors as gzip in hdf5 do not achieve strong compression for segmentation volumes. Compresso (window size 8,8,1) offers strong compression when combined with LZMA encoding, but then random access is not possible. CSGV ( $b = 64$ ) has strong compression rates, but only random access on a brick level. Neuroglancer (block size 8) encodes volumes with true random access, but has significantly worse compression rates than our method, CSGV-R ( $b = 64$ ). For H01 we evaluate a representative  $1024^3$  sub-volume based on an average gzip compressed size as not all our implementations support large chunked volumes. **Best random access and overall compression is highlighted.**

		local shading				shadow rays				ambient occlusion			
		CELLS	FIBER	H01	AZBA	CELLS	FIBER	H01	AZBA	CELLS	FIBER	H01	AZBA
CSGV rANS	brick cache	11.8	6.0	4.1	10.3	13.1	7.8	4.2	12.4	18.1	9.2	4.8	17.2
CSGV-R no stop bits	no cache	25.3	7.6	-	19.2	32.9	12.7	-	25.0	37.4	15.4	-	31.1
	voxel cache	10.1	6.4	-	9.3	12.8	9.2	-	11.6	13.2	11.9	-	13.3
	voxel cache (es)	<b>5.4</b>	5.8	-	<b>7.9</b>	<b>6.7</b>	8.1	-	<b>10.0</b>	<b>8.1</b>	10.7	-	<b>12.2</b>
	brick <sup>c</sup> cache (sm)	8.1	<b>4.5</b>	-	9.3	9.4	<b>6.4</b>	-	11.2	11.3	<b>7.4</b>	-	13.9
CSGV-R+sb	no cache	44.9	12.8	6.5	28.6	59.5	22.5	8.6	37.9	70.4	29.6	14.1	47.2
	voxel cache	9.7	6.3	<b>1.2</b>	8.8	12.4	9.0	<b>1.3</b>	10.9	13.2	11.9	<b>1.8</b>	13.0
	voxel cache (es)	6.1	7.1	75.8	8.1	7.6	9.9	79.9	10.2	9.4	13.3	109.1	12.6
	brick <sup>c</sup> cache (sm)	11.0	5.6	22.4	10.9	12.3	7.3	22.6	12.9	14.6	8.4	23.6	16.3

**Table 4:** Average milliseconds per frame when rendering a camera path (2649, 1401, 2649, 2070 frames) for CELLS, FIBER, H01, and AZBA with different cache and shading modes. H01 uses  $b = 64$ , others  $b = 32$ . The cache size is 4 GiB for H01 with CSGV and 1 GiB for others. For CSGV-R without stop bits, the compressed H01 does not fit into VRAM. CSGV uses rANS compression and only supports brick caching. With shadow rays and ambient occlusion, one secondary ray is cast per frame per pixel after the primary hit. Additional timings are in Table S1.



**Figure 8:** The latency of CSGV's brick cache creates artifacts when bricks are not yet decoded and their palette is sampled instead. Our CSGV-R always accesses correct labels.

of valid rendered samples per frame. CSGV-R with no cache or voxel caching produces an artifact-free image in the first frame. As direct voxel access is always available, the voxel cache does not have to fit all visible areas and its size can therefore be chosen freely.

## 6. Conclusion

We presented a full random access and lossless compression for segmentation volumes that outperforms the current state-of-the-art with random access in terms of compression rates. At the same time, its high query performance makes it suitable for GPU-based processing as in rendering. Our CSGV-R representation enables a

variety of voxel access schemes while the CSGV method is limited to brick caching which we outperform as well. Our method removes rendering artifacts from CSGV while improving render performance by a factor of up to 5 and allows us to render segmentation volumes with over one million labels and sizes of up to 770 GB on a consumer system. CSGV-R's limitations are weaker compression rates: while not all areas might be decoded at once, CSGV at least fits larger volumes into GPU memory. Larger  $b$  decrease performance disproportionately in CSGV-R as the deeper LOD hierarchies are traversed per voxel. CSGV remains preferred for storage and serial processing. More generally, we have shown how wavelet trees can be leveraged as a random access variable bit-length compression in high-performance rendering tasks and how they can be optimized for faster queries with domain knowledge. In future work, we want to explore how to further improve compression rates of our CSGV-R volumes, for example through methods from bit vector compression, while maintaining rendering performance with more adaptive multi-resolution empty space skipping and caching.

**Acknowledgements.** 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 the Pilot Program Core Informatics. We thank the NIC Research Group Computational Structural Biology, Jülich Research Center and the Computed Tomography group, University of Applied Sciences Upper Austria, Campus Wels, for providing data sets. Open Access funding enabled and organized by Projekt DEAL.



## References

- [AAAT\*22] AGUS M., ABOULHASSAN A., AL THELAYA K., PINTORE G., GOBBETTI E., CALÌ C., SCHNEIDER J.: Volume puzzle: visual analysis of segmented volume data with multivariate attributes. In *Proc. IEEE Visualization and Visual Analytics* (2022), pp. 130–134. doi:10.1109/VIS54862.2022.00035. 2
- [AABH\*16] AI-AWAMI A. K., BEYER J., HAEHN D., KASTHURI N., LICHTMAN J. W., PFISTER H., HADWIGER M.: Neuroblocks – visual tracking of segmentation and proofreading for large connectomics projects. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 738–746. doi:10.1109/TVCG.2015.2467441. 2
- [ATAS21] AL-THELAYA K., AGUS M., SCHNEIDER J.: The Mixture Graph – A Data Structure for Compressing, Rendering, and Querying Segmentation Histograms. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 645–655. doi:10.1109/TVCG.2020.3030451. 2
- [AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proc. Eurographics - Technical Papers* (1987), Eurographics Association. doi:10.2312/egtp.19871000. 6
- [BAAK\*13] BEYER J., AL-AWAMI A., KASTHURI N., LICHTMAN J. W., PFISTER H., HADWIGER M.: Connectomeexplorer: Query-guided visual analysis of large volumetric neuroscience data. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2868–2877. doi:10.1109/TVCG.2013.142. 2
- [BBB\*17] BOERGENS K. M., BERNING M., BOCKLISCH T., BRÄUNLEIN D., DRAWITSCH F., FROHNHOFEN J., HEROLD T., OTTO P., RZEPKA N., WERKMEISTER T., ET AL.: webKnossos: efficient on-line 3D data annotation for connectomics. *Nature Methods* 14, 7 (2017), 691–694. doi:10.1038/nmeth.4331. 2
- [BGKS15] BABENKO M. A., GAWRYCHOWSKI P., KOCIUMAKA T., STARIKOVSKAYA T.: Wavelet trees meet suffix trees. In *SODA* (2015), SIAM, pp. 572–591. doi:10.1137/1.9781611973730.39. 4
- [Ble18] BLENDER O. C.: *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. URL: <http://www.blender.org>. 2
- [BMA\*18] BEYER J., MOHAMMED H., AGUS M., AL-AWAMI A. K., PFISTER H., HADWIGER M.: 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
- [BRGIG\*14] BALSÁ RODRÍGUEZ M., GOBBETTI E., IGLESIAS GUITIÁN J., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum* 33, 6 (2014), 77–100. doi:https://doi.org/10.1111/cgfm.12280. 2
- [BSL18] BERGER D. R., SEUNG H. S., LICHTMAN J. W.: 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
- [BTB\*22] BEYER J., TROIDL J., BOORBOOR S., HADWIGER M., KAUFMAN A., PFISTER H.: A Survey of Visualization and Analysis in High-Resolution Connectomics. *Computer Graphics Forum* 41, 3 (2022), 573–607. doi:https://doi.org/10.1111/cgfm.14574. 2
- [CKV24] CEREGINI M., KURPICZ F., VENTURINI R.: Faster wavelet tree queries. In *DCC* (2024), IEEE, pp. 223–232. doi:10.1109/DCC58796.2024.00030. 3
- [CNP15] CLAUDE F., NAVARRO G., PEREIRA A. O.: The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.* 47 (2015), 15–32. doi:10.1016/j.is.2014.06.002. 3, 5, 6
- [CNS11] CLAUDE F., NICHOLSON P. K., SECO D.: Space efficient wavelet tree construction. In *SPIRE* (2011), vol. 7024 of *Lecture Notes in Computer Science*, Springer, pp. 185–196. doi:10.1007/978-3-642-24583-1\_19. 4
- [CW77] CARTER J. L., WEGMAN M. N.: Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1977), STOC '77, Association for Computing Machinery, p. 106–112. doi:10.1145/800105.803400. 7
- [DEF\*21] DINKLAGE P., ELLERT J., FISCHER J., KURPICZ F., LÖBEL M.: Practical wavelet tree construction. *ACM J. Exp. Algorithmics* 26 (2021), 1.8:1–1.8:67. doi:10.1145/3457197. 4, 5
- [dFdS17] DA FONSECA P. G. S., DA SILVA I. B. F.: Online construction of wavelet trees. In *SEA* (2017), vol. 75 of *LIPICs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:14. doi:10.4230/LIPICs.SEA.2017.16. 4
- [DFK20] DINKLAGE P., FISCHER J., KURPICZ F.: Constructing the wavelet tree and wavelet matrix in distributed memory. In *ALENEX* (2020), SIAM, pp. 214–228. doi:10.1137/1.9781611976007.17. 4
- [DFKT23] DINKLAGE P., FISCHER J., KURPICZ F., TARNOWSKI J.: Bit-parallel (compressed) wavelet tree construction. In *DCC* (2023), IEEE, pp. 81–90. doi:10.1109/DCC55655.2023.00016. 4
- [DHKP97] DIETZFELBINGER M., HAGERUP T., KATAJAINEN J., PENTTONEN M.: A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms* 25, 1 (1997), 19–51. doi:https://doi.org/10.1006/jagm.1997.0873. 7
- [Dud13] DUDA J.: Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540* (2013). doi:10.48550/arXiv.1311.2540. 2, 4
- [EHK\*06] ENGEL K., HADWIGER M., KNISS J., REZK-SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. AK Peters/CRC Press, 2006. 2
- [EK19] ELLERT J., KURPICZ F.: Parallel external memory wavelet tree and wavelet matrix construction. In *SPIRE* (2019), vol. 11811 of *Lecture Notes in Computer Science*, Springer, pp. 392–406. doi:10.1007/978-3-030-32686-9\_28. 4
- [FEFS17] FUENTES-SEPÚLVEDA J., ELEJALDE E., FERRES L., SECO D.: Parallel construction of wavelet trees on multicore architectures. *Knowl. Inf. Syst.* 51, 3 (2017), 1043–1066. doi:10.1007/s10115-016-1000-6. 4
- [FM00] FERRAGINA P., MANZINI G.: Opportunistic data structures with applications. In *FOCS* (2000), IEEE Computer Society, pp. 390–398. doi:10.1109/SFCS.2000.892127. 3
- [FMMN07] FERRAGINA P., MANZINI G., MÄKINEN V., NAVARRO G.: Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* 3, 2 (2007), 20. doi:10.1145/1240233.1240243. 3
- [GG92] GRANER F., GLAZIER J. A.: Simulation of biological cell sorting using a two-dimensional extended pots model. *Physical review letters* 69, 13 (1992), 2013. doi:10.1103/PhysRevLett.69.2013. 8
- [GGV03] GROSSI R., GUPTA A., VITTER J. S.: High-order entropy-compressed text indexes. In *SODA* (2003), ACM/SIAM, pp. 841–850. 3
- [GNP20] GAGIE T., NAVARRO G., PREZZA N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM* 67, 1 (2020), 2:1–2:54. doi:10.1145/3375890. 3
- [Goo16] GOOGLE INC.: Neuroglancer. [github.com/google/neuroglancer](https://github.com/google/neuroglancer), 2016. 2, 8
- [GVX11] GROSSI R., VITTER J. S., XU B.: Wavelet trees: From theory to practice. In *CCP* (2011), IEEE Computer Society, pp. 210–221. doi:10.1109/CCP.2011.16. 3
- [HBG\*24] HONG A., BOUCHER C., GAGIE T., LI Y., ZEH N.: Another virtue of wavelet forests. In *SPIRE* (2024), vol. 14899 of *Lecture Notes in Computer Science*, Springer, pp. 184–191. 3
- [HDF] HDF GROUP, THE: Hierarchical Data Format, version 5. <https://github.com/HDFGroup/hdf5>. 2, 8

- [Huf52] HUFFMAN D. A.: A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. doi:10.1109/JRPROC.1952.273898. 4
- [Jac89] JACOBSON G.: Space-efficient static trees and graphs. In *FOCS* (1989), pp. 549–554. doi:10.1109/SFCS.1989.63533. 3
- [JO20] JARZYNSKI M., OLANO M.: Hash functions for gpu rendering. *Journal of Computer Graphics Techniques (JCGT)* 9, 3 (October 2020), 20–38. 7
- [Kan18] KANETA Y.: Fast wavelet tree construction in practice. In *SPIRE* (2018), vol. 11147 of *Lecture Notes in Computer Science*, Springer, pp. 218–232. doi:10.1007/978-3-030-00479-8\_18. 4
- [KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel dags. *ACM Transactions on Graphics* 32, 4 (7 2013). doi:10.1145/2461912.2462024. 2
- [KSY\*21] KENNEY J. W., STEADMAN P. E., YOUNG O., SHI M. T., POLANCO M., DUBAISHI S., COVERT K., MUELLER T., FRANKLAND P. W.: A 3d adult zebrafish brain atlas (azba) for the digital age. *eLife* 10 (11 2021), e69988. doi:10.7554/eLife.69988. 8
- [Kur22] KURPICZ F.: Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE* (2022), vol. 13617 of *Lecture Notes in Computer Science*, Springer, pp. 257–272. doi:10.1007/978-3-031-20643-6\_19. 3, 5, 6
- [LAB\*24] LESAR Ž., ALHARBI R., BOHAK C., STRNAD O., HEINZL C., MAROLT M., VIOLA I.: Volume conductor: interactive visibility management for crowded volumes. *The Visual Computer* 40, 2 (2024), 1005–1020. 2
- [LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *Proc. ACM SIGGRAPH* 21, 4 (8 1987), 163–169. doi:10.1145/37402.37422. 2
- [Lem10] LEMPITSKY V.: Surface extraction from binary volumes with higher-order smoothness. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2010), pp. 1197–1204. doi:10.1109/CVPR.2010.5539832. 2
- [LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, p. 55–63. doi:10.1145/1730804.1730814. 2
- [LM14] LAROBINA M., MURINO L.: Medical image file formats. *Journal of digital imaging* 27 (2014), 200–206. 2
- [MAF07] MORELAND K., AVILA L., FISK L. A.: Parallel unstructured volume rendering in ParaView. In *Visualization and Data Analysis 2007* (2007), Erbacher R. F., Roberts J. C., Gröhn M. T., Börner K., (Eds.), vol. 6495, International Society for Optics and Photonics, SPIE, p. 64950F. doi:10.1117/12.704533. 2
- [Mak12] MAKIS C.: Wavelet trees: A survey. *Comput. Sci. Inf. Syst.* 9, 2 (2012), 585–625. doi:10.2298/CSIS110606004M. 3
- [MHL\*17] MATEJEK B., HAEHN D., LEKSCHAS F., MITZENMACHER M., PFISTER H.: Compresso: Efficient Compression of Segmentation Data For Connectomics. In *Medical Image Computing and Computer-Assisted Intervention* (Cham, 2017), Springer, pp. 781–788. doi:10.1007/978-3-319-66182-7\_89. 2
- [MJB\*21] MINNEN D., JANUSZEWSKI M., BLAKELY T., SHAPSON-COE A., SCHALEK R. L., BALLÉ J., LICHTMAN J. W., JAIN V.: Denoising-based image compression for connectomics. *bioRxiv* (2021). doi:10.1101/2021.05.29.445828. 2
- [MN06] MÄKINEN V., NAVARRO G.: Position-restricted substring searching. In *LATIN* (2006), vol. 3887 of *Lecture Notes in Computer Science*, Springer, pp. 703–714. 3
- [MNV16] MUNRO J. I., NEKRICH Y., VITTER J. S.: Fast construction of wavelet trees. *Theor. Comput. Sci.* 638 (2016), 91–97. doi:10.1016/j.tcs.2015.11.011. 4
- [MSJ\*22] MAURER J., SALABERGER D., JERABEK M., KASTNER J., MAJOR Z.: 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 (2022), 582–600. doi:10.1080/10589759.2022.2075865. 8
- [MSL22] MAITIN-SHEPARD J., LEAVITT L.: TensorStore for High-Performance, Scalable Array Storage. <https://research.google/blog/tensorstore-for-high-performance-scalable-array-storage/>, 2022. 2
- [MVD\*18] MUELLER J. H., VOGLREITER P., DOKTER M., NEFF T., MAKAR M., STEINBERGER M., SCHMALSTIEG D.: Shading atlas streaming. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 37, 6 (Dec. 2018). doi:10.1145/3272127.3275087. 6, 9
- [NPS20] NVIDIA CORPORATION, PANTALEONI J., SUBTIL N.: NVBIO. <https://nvlabs.github.io/nvbio/>, 2020. 4
- [O’N14] O’NEILL M. E.: *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 09 2014. URL: <https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf>. 7
- [PD24] PIOCHOWIAK M., DACHSBACHER C.: Fast compressed segmentation volumes for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics* 30, 1 (2024), 12–22. doi:10.1109/TVCG.2023.3326573. 1, 2, 4, 5, 7, 8, 9
- [PHK04] PIEPER S., HALLE M., KIKINIS R.: 3d slicer. In *2004 2nd IEEE international symposium on biomedical imaging: nano to macro (IEEE Cat No. 04EX821)* (2004), IEEE, pp. 632–635. 2
- [PR04] PAGH R., RODLER F. F.: Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. doi:https://doi.org/10.1016/j.jalgor.2003.12.002. 7
- [QDB11] QUEY R., DAWSON P., BARBE F.: Large-scale 3d random polycrystals for the finite element method: Generation, meshing and remeshing. *Computer Methods in Applied Mechanics and Engineering* 200 (04 2011), 1729–1745. doi:10.1016/j.cma.2011.01.002. 2
- [RBS20] ROSENBAUER J., BERGHOFF M., SCHUG A.: Emerging tumor development by simulating single-cell events. *bioRxiv* (2020). doi:10.1101/2020.08.24.264150. 8
- [RP71] RICE R., PLAUNT J.: Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology* 19, 6 (1971), 889–897. doi:10.1109/TCOM.1971.1090789. 5
- [SCJB\*24] SHAPSON-COE A., JANUSZEWSKI M., BERGER D. R., POPE A., WU Y., BLAKELY T., SCHALEK R. L., LI P. H., WANG S., MAITIN-SHEPARD J., KARLUPA N., DORKENWALD S., SJOSTEDT E., LEAVITT L., LEE D., TROIDL J., COLLMAN F., BAILEY L., FITZMAURICE A., KAR R., FIELD B., WU H., WAGNER-CARENA J., ALEY D., LAU J., LIN Z., WEI D., PFISTER H., PELEG A., JAIN V., LICHTMAN J. W.: A petavoxel fragment of human cerebral cortex reconstructed at nanoscale resolution. *Science* 384, 6696 (2024), eadk4858. doi:10.1126/science.adk4858. 8
- [Shu15] SHUN J.: Parallel wavelet tree construction. In *DCC* (2015), IEEE, pp. 63–72. doi:10.1109/DCC.2015.7. 4
- [Shu20] SHUN J.: Improved parallel construction of wavelet trees and rank/select structures. *Inf. Comput.* 273 (2020), 104516. doi:10.1016/j.ic.2020.104516. 4
- [Tis11] TISCHLER G.: On wavelet tree construction. In *CPM* (2011), vol. 6661 of *Lecture Notes in Computer Science*, Springer, pp. 208–218. doi:10.1007/978-3-642-21458-5\_19. 4
- [VMM\*23] VELICKY P., MIGUEL E., MICHALSKA J. M., LYUDCHIK J., WEI D., LIN Z., WATSON J. F., TROIDL J., BEYER J., BEN-SIMON Y., ET AL.: Dense 4d nanoscale reconstruction of living brain tissue. *Nature Methods* 20, 8 (2023), 1256–1265. 2
- [WPD24] WERNER M., PIOCHOWIAK M., DACHSBACHER C.: SVDAG Compression for Segmentation Volume Path Tracing. In *Vision, Modeling, and Visualization* (2024), Linsen L., Thies J., (Eds.), The Eurographics Association. doi:10.2312/vmv.20241196. 2