STAG: Smart Tools and Applications in Graphics (2019)
M. Agus, M. Corsini and R. Pintus (Editors)

MTV-Player: Interactive Spatio-Temporal Exploration of Compressed Large-Scale Time-Varying Rectilinear Scalar Volumes

J. Díaz[1], F. Marton[2], and E. Gobbetti[2]

[1]Digital Care Research Group, UVic-UCC, Spain
[2]Visual Computing Group, CRS4, Italy

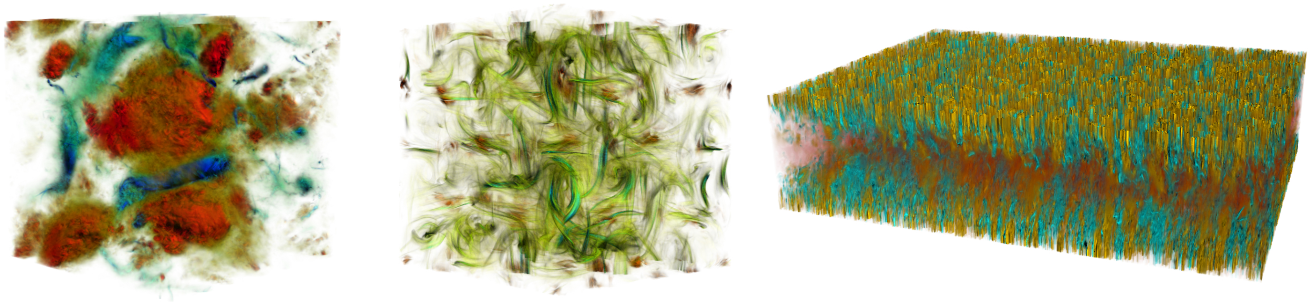

Figure 1: ***Real-time exploration.*** *Our multiresolution compression-domain GPU volume rendering architecture supports interactive random-access exploration of massive time-varying rectilinear scalar volumes on commodity platforms. From left to right:* 1024^4 *Isotropic Turbulence simulation (4 TB) compressed to 70.1 GB (0.34 bps, PSNR 67.9 dB);* 1024^4 *Forced MHD Turbulence simulation (4 TB) compressed to 70.7 GB (0.34 bps, PSNR 52.51 dB).* $(2048 \times 512 \times 1536) \times 4000$ *Channel Flow simulation (23.4 TB) compressed to 416.8 GB (0.34 bps, PSNR 57.51 dB).*

Abstract

We present an approach for supporting fully interactive exploration of massive time-varying rectilinear scalar volumes on commodity platforms. We decompose each frame into a forest of bricked octrees. Each brick is further subdivided into smaller blocks, which are compactly approximated by quantized variable-length sparse linear combinations of prototype blocks stored in a data-dependent dictionary learned from the input sequence. This variable bit-rate compact representation, obtained through a tolerance-driven learning and approximation process, is stored in a GPU-friendly format that supports direct adaptive streaming to the GPU with spatial and temporal random access. An adaptive compression-domain renderer closely coordinates off-line data selection, streaming, decompression, and rendering. The resulting system provides total control over the spatial and temporal dimensions of the data, supporting the same exploration metaphor as traditional video players. Since we employ a highly compressed representation, the bandwidth provided by current commodity platforms proves sufficient to fully stream and render dynamic representations without relying on partial updates, thus avoiding any unwanted dynamic effects introduced by current incremental loading approaches. Moreover, our variable-rate encoding based on sparse representations provides high-quality approximations, while offering real-time decoding and rendering performance. The quality and performance of our approach is demonstrated on massive time-varying datasets at the terascale, which are nonlinearly explored at interactive rates on a commodity graphics PC.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation—Computer Graphics [I.3.7]: Three-dimensional graphics and realism—Coding and Information Theory [E.4]: Data compaction and compression—Compression (Coding) [I.4.2]: Approximate methods—

1. Introduction

Interactive visual exploration of very large time-varying datasets is crucial to understand scientific simulation results [WF08, SBN11]. One very common representation is the sequence of time-varying rectilinear scalar volumes. Such data routinely has thousands of time steps and billions of voxels per frame [LPW*08, Iri06]. Meeting interactivity constraints when rendering such data is very hard, especially when showing animation. In order to cope with bandwidth constraints, all current GPU-based solutions mix and match multiresolution data representations, compression, out-of-core methods and data streaming to enable the interactive visualization of massive volumetric datasets. With the notable exception of the adaptive

DOI: 10.2312/stag.20191358

variable-rate approach of Marton et al. [MAG19], however, all current solutions either amortize the updates of a rendering working-set over multiple frames, introducing unwanted dynamic effects, or use differential encodings which restrict random access and non-trivial backward/forward/accelerated temporal exploration of time-varying sequences (see Sec. 2).

In this paper, we explore the feasibility of adapting state-of-the-art compressed octree-based solutions to offer spatio-temporal random access without incremental updates. The resulting system strives to provide total control over the spatial and temporal dimensions of the data, supporting the same exploration metaphor as traditional video players (hence the name *MTV-Player* for *Massive Time-varying Volume Player*).

By encoding frames in a I/O and GPU-friendly compressed format supporting high-compression rate with tolerance-driven error control, we strive to stream to GPU and render dynamic scenes with 1Gvoxel/frame. To achieve this goal, each frame is independently encoded, and decomposed into a forest of bricked octrees, which are used as culling and I/O units. Each brick is further subdivided into smaller blocks, which are compactly approximated by variable-length sparse linear combinations of prototype blocks stored in a data-dependent dictionary learned from the input sequence (Sec. 3). At run-time, an adaptive compression-domain renderer closely coordinates off-line data selection, streaming, decompression, and rendering, starting from an out-of-core GPU-friendly representation that supports adaptive streaming to the GPU with spatio-temporal random access (Sec. 4).

Our contributions are manifold. First, we introduce a carefully designed I/O and GPU-friendly compact representation. Second, we show how a tolerance-driven variable-rate codec scheme can support scalable dictionary learning on massive datasets. Third, we show how our variable-rate encoding based on sparse representations provides scalable high-quality approximations, while offering real-time transient decoding within an interactive renderer. Finally, we describe how the codec can be integrated in an out-of-core real-time rendering capable to fully stream and render dynamic representations without relying on partial updates. The major limitations of the method, shared with other compression-based approaches, are the non-negligible encoding time and the upper bound on achievable quality dictated by the need to meet run-time bandwidth constraints for giga-voxel-sizes working sets. Our results show, however, that excellent quality results can be achieved on time-varying datasets with billions of voxels per frame and thousands of time-steps, making the method of immediate practical interest.

2. Related Work

This section briefly reviews the work that is most closely related to our approach. For additional information, we refer the reader to the following surveys on modeling and visualization methods for time-varying volumetric data [WF08], compression-based direct volume rendering [BRGIG*14] and GPU-based large-scale volume visualization [BHP15].

Data compression for GPU-accelerated direct volume rendering State-of-the-art solutions are based on compressing and storing data in multiresolution out-of-core structures such as octrees [CNLE09, Eng11, GIM12, RTW13], bricks [TBR*12], hierarchical grids of bricks [HBJP12, FSK13] or hierarchical tiled 3D grids [MAG19], followed by an adaptive loading of the compressed data on the GPU, where a fast decompression is performed on-demand during rendering. In particular, sustaining a 10 frames/s animation on $1K^3$ working sets requires uploading, decompressing, and rendering at least 10 Gvox/s. To this end, a wide variety of compression methods have been used. Early successful approaches used simple hardware-accelerated fixed-rate codecs [Cra, YNV08, IGM10] that allow random access, but limit achievable compression and rate-distortion performance [FM07, PK09]. Vector quantization solutions [SW03, KE02, GG16, YZW*17] have also demonstrated real-time performance, but their quality is limited by dictionary size [AEB06]. Several more advanced codecs, in particular based on Wavelets [Lin14, AFdMSP18] and tensor approximation solutions [SIM*11, BRLP18] have demonstrated the capability to achieve excellent performance, especially when combined with advanced entropy coding methods. Due to their parallel decoding complexity, at low bit rates, even the fastest methods [TBR*12] are far from being able to sustain real-time streaming of large dynamic volumes on current hardware generation [MAG19]. For this reason, interactive systems are typically forced to amortize decompression over multiple frames, which is not a suitable solution for time-varying data [MAG19]. Park et al. [PGEK17] have proposed to improve performance by proposing a saliency-aware codec, in order to distribute coding efforts into areas deemed more important, but their approach does not allow changes of the transfer function on-the-fly. Sparse-coding methods, which represent volume blocks as sparse linear combinations of prototype blocks from a learned overcomplete dictionary, have shown to achieve state-of-the-art compression while supporting real-time decoding, as demonstrated by the COVRA architecture [GIM12]. As the dictionary is learned from the data, the number of terms needed for good approximation at low bit rates is much lower than with fixed bases such as wavelets, so that state-of-the-art rate-distortion performance can be achieved without recurring to complex entropy coding methods. Tensor decomposition is also learned from data, but it imposes limitations on decoding locality and achieves a much lower decoding speed [SIM*11, BRLP18]. COVRA, however, used a fixed-rate approach, which is sub-optimal for datasets that exhibit wide spatial variations in data complexity, a common feature in many simulations. More recently, Marton et al. [MAG19] have proposed a variation which improve over COVRA's fixed-rate scheme through a constrained variable-rate encoder which adapts bit rates within fixed-size pages, as well as with a better bit allocation scheme. In this work, we show how a tolerance-driven approach leading to an unconstrained variable-rate encoding can significantly improve over fixed-rate schemes and provide also superior results in terms of rate-distortion with respect to page-constrained schemes. As the drawback of these methods is the lengthy dictionary learning time, we also propose a faster training scheme based on a hierarchy of coresets.

Time-varying data exploration Visualizing time-varying datasets has usually been addressed with temporal-coherent compression techniques [SJ94, GS01, LMC02, WWS03, WWS*05], where data of previous time-steps are needed to process a specific one. This extra amount of information imposes a rigid constraint in terms of memory and bandwidth usage that limits the maximum size of the input data and difficult random access to specific time-steps. Thus, the free temporal exploration like the one provided by traditional video players that we seek with our approach is difficult to be obtained. Random access to single time-steps is improved by compressing voxels with respect to reference key-frames [Wes95, MS00, WGLS05, She06, KLW*08, MRH10, WYM08, SBN11, JEG12], but restrictions on the maximum size of data are still present, even when combining both approaches [FM07, NIH08, WYM10, CWW11]. A different way to partially overcome the limitations of the temporal-coherent compression architectures is based on encoding each frame of the sequence individually by using 3D compression methods [GIM12, TBR*12, PLK*18, MAG19]. With no temporal dependency, full random access to single time-steps is guaranteed. As in the COVRA architecture [GIM12], our technique encodes volume blocks using a sparse representation based on a dictionary learned by means of the K-SVD algorithm [AEB06]. However, instead of using a fixed-rate compression scheme, our variable-rate encoding provides a better tradeoff between quality and compression ratios. Moreover, COVRA relies on incremental updates, and dynamic datasets are visualized only by fully pre-caching dynamic data on the GPU, while we support full-frame updates and unlimited-length sequences. More examples of per-frame based approaches are the wavelet-based compression rendering architecture presented in [TBR*12], which includes run-length and entropy encoding, or the recent method by Pulido et al. [PLK*18] that allows the remote visualization of multi-terabyte data employing as well a wavelet-based compression scheme. High quality explorations of single time-steps are obtained in both cases but a free temporal exploration of the sequence can not be achieved in real-time. The recent work of Marton et al. [MAG19] also supports high-quality exploration of massive time-varying datasets in both desktop and mobile devices using a variable-rate encoding scheme based on sparse coding with fixed-size pages, using a hierarchy of grids. This feature allows a fast parallel decompression on the GPU but it is not well-suited for datasets with big empty regions, where many pages might not be completely filled, producing a non-optimal memory footprint of the compressed data. To address this issue, we employ here a sparse octree representation with variable sized bricks.

3. Constructing the compact multiresolution out-of-core representation

Our method is based on the off-line transformation of a time-varying rectilinear scalar volume into a compressed representation stored off-line in a format that supports random access to individual frames, quick coarse determination of the portions required for a particular image, efficient streaming to GPU and rendering of those portions. As illustrated in Fig. 2, each time step is spatially split into subvolumes of the same size, which constitute the unit for coarse culling, I/O and rendering. For finer level culling and compression, a multiresolution hierarchy of bricks is computed for each subvolume, compressed and stored in an octree with a GPU-friendly format that can be transferred from GPU to disk using batched asynchronous host to device copies (see Sec. 4).

In order to build the compact representation, we choose an error-driven approach in which each brick of size B is further subdivided into smaller blocks of size M, which are compactly approximated

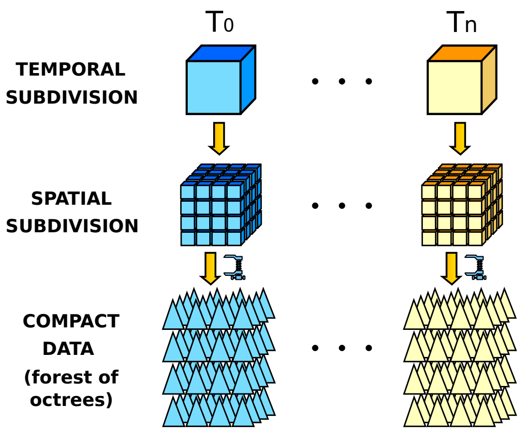

Figure 2: ***Compact representation of the data.*** *First, the input sequence is temporally subdivided into the individual time-steps (T_0, ..., T_n). Then, each one is spatially split into subvolumes of the same size, which will be used for coarse culling and as I/O units to move data from disk to GPU. Finally, a multiresolution hierarchy is computed for each subvolume, compressed and stored in an octree with a GPU-friendly format that can be culled and selectively transferred from GPU to disk using batched asynchronous host-to-device copies.*

up to a prescribed error tolerance by sparse linear combinations of prototype blocks stored in a data-dependent dictionary $\mathbf{D}$, learned from the entire sequence using a process driven by the same error tolerance used for the encoding. Such an adaptive approach overcomes the problems of fixed-rate schemes, which often lead to poor reconstructions of high-variation regions. All the computation is performed in a scalable way, without limits dictated by the input data size. The parameters guiding the process are the brick size B, which determines the octree granularity, the compressed block size $M \leq B$, the desired coreset size C, which bounds the amount of memory used for training, the dictionary size $K \geq M^3$, and a threshold $\varepsilon \geq 0$ to bound the reconstruction error of each block.

3.1. Dictionary learning

In our approach, an over-complete dictionary $\mathbf{D}$ of prototype blocks is trained using a tolerance-driven method from the input data and each block b_i of the original volume is represented by a sparse linear combination of prototype blocks. To do this, each block b_i of size $m = M^3$ is first mapped to a zero-mean column vector $\mathbf{y} \in \mathbb{R}^m$ by subtracting its average $\bar{y}$. Since we target error-constrained variable-rate compression, the dictionary is computed by jointly optimizing the columns of the dictionary and the sparse representation given the error tolerance ε according to the objective function:

$$\min_{\mathbf{D},\lambda_i} \sum_i w_i \|\mathbf{y}_i - \mathbf{D}\lambda_i\|_2^2 \qquad (1)$$

subject to:

$$\forall i, \|\lambda_i\|_0 \text{ is the minimum such that } \|\mathbf{y}_i - \mathbf{D}\lambda_i\|_2^2 \leq \varepsilon^2 \qquad (2)$$

where w_i represents the weight of each training sample and $\|\lambda_i\|_0$ is the number of non-zero entries of λ_i. A variation of the K-SVD algorithm [AEB06] is used to train the dictionary $\mathbf{D}$, where each prototype block is mapped to a unitary column vector $\mathbf{d}_i \in \mathbb{R}^m$. All $\mathbf{d}_i$ are initialized randomly and maintained at zero mean.

The K-SVD method alternatively iterates between two subproblems solved by heuristic greedy methods: sparse coding, which finds the best λ_i given a fixed dictionary, and dictionary updating. For sparse coding, we employ the batch-OMP algorithm [RZE08], while for dictionary updating we employ the Approximate K-SVD algorithm [RZE08], which we modified to take into account training sample weights, similarly to previous work [GIM12, MAG19], in order to reduce the complexity of dictionary learning by performing it on a weighted subset of the original samples (i.e., a coreset) instead of on all the input samples.

Instead of building a single coreset of size C, we extract a hierarchy of coresets at different resolutions $C_n = C, C_{n-1} = \frac{C}{2}, C_{n-2} = \frac{C}{4}, \ldots, C_0$, where n is the number of sub-sampling levels (6 in this paper). Using this hierarchy of coresets, training is performed with the iterative refinement process shown in Algorithm 1, which applies the iterative K-SVD method to larger and larger coresets, in contrast to previous work [GIM12, MAG19], that uses a non-hierarchical approach. Using coresets of reduced size in the first loops and larger ones as refinement proceeds makes it possible to have the dictionary quickly converge to a coarse but valid solution and to concentrate efforts on the last fine-tuning steps, improving convergence and accelerating the training process.

It is important to note that extracting a coreset hierarchy does not require multiple streaming passes, as multiple coresets can be obtained by generating multiple streams in parallel at different sampling rates. Moreover, in order to further reduce I/O bottleneck, we generate coresets by randomly selecting a number of frames F in the input sequence using Halton sampling for the time dimension until the number of blocks contained in the selected frame set is significantly larger than the coreset size (16 times in this paper), and only construct the coreset from these frames.

Algorithm 1 Dictionary Training

```
1:  Input: Coreset hierarchy $C$, tolerance $\varepsilon$, refinement loops $N$
2:  Output: Overcomplete dictionary of prototype blocks $\mathbf{D}$
3:  $\mathbf{D} \leftarrow [\mathbf{1}, \mathbf{random}, \ldots, \mathbf{random}]$
4:  for $l = 1$ to max_level($C$) do
5:      $\tilde{\mathbf{Y}} \leftarrow$ blocks($C_l$)
6:      $\tilde{\mathbf{W}} \leftarrow$ weights($C_l$)
7:      for $i = 0$ to N do
8:          // Dictionary refinement
9:          $\lambda \leftarrow$ batch_OMP($\tilde{\mathbf{Y}}$, $\varepsilon$)
10:         $\mathbf{D} \leftarrow$ update_dictionary_entries($\mathbf{D}$, $\tilde{\mathbf{W}}$, $\lambda$)
11:     end for
12: end for
```

3.2. Dataset encoding

Given the computed dictionary $\mathbf{D}$, each frame is transformed, in parallel, into a forest of octrees, such that the number of octrees is no less than a predefined number (64 in this paper), and that each octree, once uncompressed, has a footprint smaller than the maximum transient GPU memory required for rendering (256MB for the examples in this paper). Having a forest of small octrees simplifies coarse-grained culling and memory management in the run-time phase (see Sec. 4). The encoding process is massively

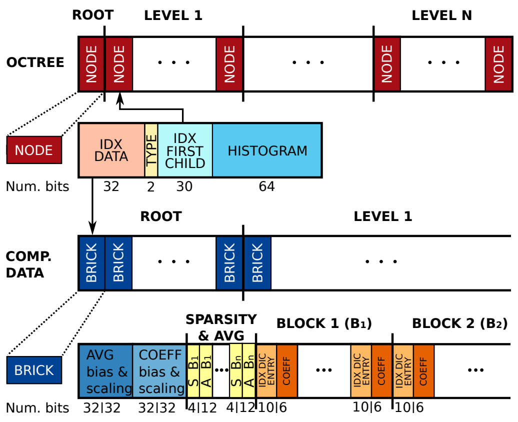

Figure 3: ***Encoded format.*** *Each octree is stored in breadth-first order (and Morton order inside each level) to support efficient partial GPU uploading up to a run-time-determined levels. The first data segment stores the octree-structure using 128 bits per non-empty node, while the second data segment stores a variable-bit-rate representation of each non-empty brick, composed of a header plus a quantized sparse representation of its blocks.*

parallel and requires minimum memory, as it can be performed in parallel for all bricks that compose each octree.

The off-line storage of each octree of bricks, illustrated in Fig. 3, is designed to be compact, GPU-decodable, and whose visible portion can be directly transferable to the GPU up to the required levels using few batched host to device copies (see Sec. 4). The representation consists of two parts: the octree structure itself, and the data for the non-empty bricks.

In the octree structure portion, each node is encoded in 128 bits containing an index to its compressed data brick (32 bits), the node type (2 bits: inner, leaf or empty node), the index of the first child in the nodes list (30 bits) and a binary histogram (64 bits) computed bottom-up from the bricks of the children, that is used for transfer-function-based culling. In case of leaf or empty nodes, the index to the first child is set to 0. The compressed data is encoded as a list of bricks sorted by levels.

In the data portion, non-empty bricks are compactly stored without gaps in breadth-first order (and Morton order inside each level), to make it possible to rapidly move from disk to the GPU only a part of the data when the maximum resolution is not required. The data storage for non-empty bricks contains the sparse representation of the contained blocks, which is computed by the same batch-OMP approach used in the dictionary encoding. Compression is achieved by storing for each block only the index and value of the non-empty coefficients λ_i and by quantizing their values, as well as the average value. Each brick encoding thus contains two float values (32 bits each) representing the bias and scaling for dequantizing the average value $\bar{y}$ of each contained block, and other two for the coefficients λ_i of all blocks. The dequantization header is followed by two data segments. In the first segment, which contains one entry per encoded block, we encode a pair with the sparsity s_i of each block and the quantized average of the block. The sparsity (i.e., the number of non-zero coefficients) uses 4 bits/block for a maximum sparsity of $S_{max} = 15$, while the average uses 12 bits/block. In the second segment we encode each block as a sequence s_i (16-bit codes), with the upper 10 bits providing the index of the prototype block in the dictionary, and the lower 6 bits dedicated to the corresponding quantized coefficient value. This encoding supports dictionaries with 1024 entries, which have proved adequate in all our tests. It should be noted that the maximum sparsity constraint is easily achieved by limiting the number of iterations in the batch-OMP sparse coding step for dictionary learning and dataset encoding.

4. GPU accelerated rendering

In this paper, we strive to demonstrate the feasibility of a novel compression-domain out-of-core DVR approach supporting full spatio-temporal random access without incremental updates. We therefore designed an adaptive renderer built around on-demand streaming to GPU of compressed frame portions. By design, the renderer does not use any form of temporal coherence and does not use key-framing. We assume that data is stored offline locally on SSDs, and we exploit their performance in a GPU renderer that transfers memory mapped data to the GPU at each rendering asynchronously and in very few batches. This makes it possible to quickly and freely achieve random access and non-trivial backward/forward/accelerated temporal exploration of time-varying sequences.

4.1. Adaptive loading and rendering scheme

The process followed to render a single time-step is shown in Algorithm 2.

Algorithm 2 Process to render a single time-step

```
 1: Input: Time $t$, view $V$, projection $P$, transfer function $TF$, dictionary $D$
 2: Output: Updated frame buffer
 3: clear_frame_buffer()
 4: $\mathcal{O}_t \leftarrow$ Forest of octrees of frame $t$
 5: for each subtree $o_i \in \mathcal{O}_t$ in front-to-back-order do
 6:     if is_visible($o_i, V, P, TF$) then
 7:         $(idx, data\_ranges) \leftarrow$ adaptive_selection($o_i, V, P, TF$)
 8:         $(idx, data\_ranges) \leftarrow$ merge_batches($idx, data\_ranges$)
 9:         async_host_to_device_memcpy($idx$)
10:         async_host_to_device_memcpy($data[data\_ranges]$)
11:         $dec\_bricks\_tex3D \leftarrow$ gpu_decoded($leafs, D$)
12:         gpu_ray_casting_and_compositing($dec\_bricks\_tex3D$)
13:     end if
14: end for
```

As each frame is encoded independently into a forest of octrees of bricks, we traverse front-to back each octree root of the current time-step using a simple grid ordering and determine whether the octree is potentially visible from the current view using the current transfer function parameters. This step requires only the computed bounding box and the stored value histogram of the root node. If the octree is proved invisible, it is skipped and rendering proceeds by evaluating the next one in front-to-back order. Otherwise, we prepare data transfer by computing, from the index data and the current viewing configuration, a spatial index of the data required for rendering the octree. We perform this phase within a recursive traversal of the index tree, collecting into a small array a compact spatial index of the potentially visible portion of the tree, whose leaf nodes point to associated data bricks. During this traversal, we maintain the minimum and maximum index of the data bricks per level, which determine the portions of the data array that are required for rendering. As the data array is stored in breadth-first order (and Morton order inside each level), this range efficiently culls out the data that is too refined or out-of-frustum. After collection is completed, we reduce the number of disjoint data ranges to a maximum of three, by iteratively merging nearby data ranges of different levels starting from the smaller gaps. Merging stops when the number of ranges is less than four and the next removed gap is larger than 25% of the size of the merged data ranges. At the end of merging, the different data ranges are assumed to be relocated contiguously, and the references in the index tree are updated accordingly. Rendering can then be performed by moving the index tree and the compressed brick data to GPU using few `async_host_to_device_memcpy` calls (up to a maximum of four per octree, including the index). It should be noted, in addition, that by using a different CUDA stream per subtree, we can effectively obtain concurrency and overlap between transfer and computation.

Once the data is in device memory, we decompress all volume bricks covered by the subtree into 3D texture memory using a fast CUDA-based GPU decoder (see Sec. 4.2). Decompressed data is then accessed by a GPU ray-casting algorithm, that traverses the current spatial index using standard stackless raycasting schemes [CNLE09, GIM12] and samples voxel values from the temporary decoded 3D texture with hardware texture filtering. The raycaster is implemented in a single CUDA kernel, which renders the octree to a viewport of the frame buffer that strictly encloses the projection of the octree's bounding box. The octree raycasting procedure starts from the color and opacity fetched from the frame buffer, and follows the ray accumulating colors and opacity until maximum opacity is achieved or the ray leaves the subtree. Since octrees are rendered in front-to-back order, this approach supports visibility culling through early ray termination. After rendering all the visible octrees in a frame, the frame-buffer contains the final composited image for the volume.

4.2. GPU decompression

The GPU decompression phase must transform our variable-rate representation of bricks into uncompressed data stored in a 3D texture. This is achieved through a combination of several CUDA kernels. Given the fact that bricks and blocks have a variable size, we opted for a linear GPU memory layout for compressed data, which provides an easy sequential access to the brick's data, given the availability of offsets to the start point of each brick. Before starting GPU decompression, the CPU renderer uploads to the GPU three buffers: the compressed data representation, the starting offsets of the compressed bricks and their corresponding destination positions in the 3D texture. Decompressing a block requires to know where it starts. Each brick has a header containing the sparsity of all its encoded blocks, thus before decompression all the starting positions of the blocks are evaluated with a prefix-sum algorithm. The kernel assumes that all the bricks are layered along the X axis, one after another, and the kernel grid reflects this brick distribution, see Fig. 4. The grid kernel blocks have the same size of the compressed blocks, and the threads within each block cooperate using shared memory to decode two compressed blocks at a time. The kernel work is subdivided in two stages: fetching data to shared memory and

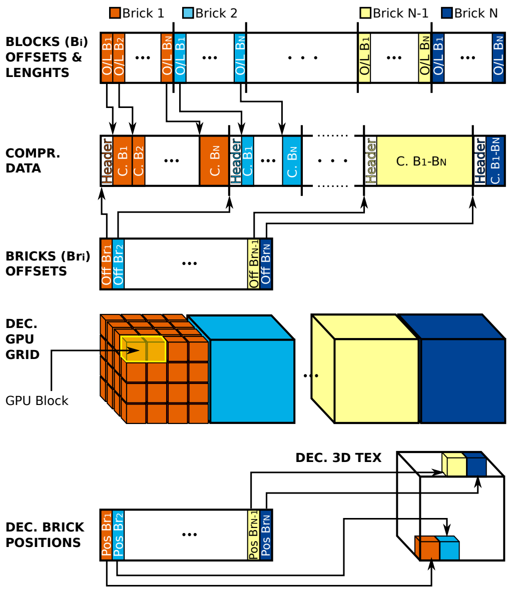

Figure 4: ***Memory layout for GPU decoding.***

voxel computation. The brick id is identified by the thread position along X, while the block id depends on the x, y, z thread coordinates. From these coordinates, the brick and block starting offsets and the block sparsity S can be loaded from the corresponding buffers. The compressed block representation (S pairs of bytes containing index and coefficient) is then cooperatively fetched and stored in shared memory. Decompressing a voxel is just a matter of loading its corresponding values from the indexed dictionary words and linearly combining them with the coefficients present in shared memory. Due to the fact that this compact representation is valid for all the voxels of the block, two adjacent voxels along X can be decoded using the same index-coefficient pairs, and can be written to output memory with a single write-4-bytes operation, since each voxel is stored in two bytes in the output texture (as we use a 16-bit format for rendering). Decoding two voxels per thread just involves loading two adjacent compressed block representations within the shared memory of a kernel block, but improves decoding performance.

5. Implementation and results

Our approach has been implemented as an experimental software running on Arch Linux using C++, OpenGL and NVIDIA CUDA 7.5. It has been tested with a variety of time-varying massive volumetric datasets. In this paper, we show the results obtained with a Forced Isotropic Turbulence (*IsoTur*) simulation (pressure field 1024^4, float: 4 TB), a Forced MHD Turbulence (*MHD*) simulation (pressure field, 1024^4, float: 4 TB) and a Channel Flow (*Channel*) simulation (pressure field, $(2048 \times 512 \times 1536) \times 4000$, float: 23.4 TB). All benchmark datasets are publicly available courtesy of the Johns Hopkins Turbulence Database (JHTDB) initiative [LPW*08].

5.1. Dictionary learning and encoding performance

We learned dictionaries for the three datasets on a PC with an Intel Core i7-3820 CPU @ 3.60 GHz and 32 GB RAM. The PC is equipped with two Samsung 840 EVO SSDs 500 GB mounted in RAID0 and with an ext4 file system, for a total available capacity of 871 GB.

In all our tests, we used a K-SVD block size $M = 6$, a brick size of $B = 32$ voxels, and a dictionary size $K = 1024$. We experimentally determined that a coreset size of $C = 256$ *MVoxels* is sufficient for learning high-quality dictionaries, and that increasing the coreset size does not substantially increase compression quality, while increasing learning time. For this reason, all the results in this paper have been computed with $C = 256$ *MVoxels*. For each dataset, 16 frames were selected using Halton sampling in the time dimension. The coresets have then been built by randomly selecting from these frames a reduced number of blocks. The coresets for the different datasets contain 17.2 *GVoxels* for IsoTur and MHD, and 26.8 *GVoxels* for Channel. Extracting coresets from data on the SSD took about 15 minutes for the first two datasets, and about 23 minutes for the last one, without any difference if extracting only a single coreset or a full coreset hierarchy, since time is dominated by the streaming passes on the input. Dictionary training has been performed using 100 iterations for the tolerances presented in Table 1, which have been selected to produce per-frame dataset sizes compatible with real-time streaming performance. Training times are independent of the dataset sizes, since they are performed on coresets. When training using the large coreset the times range from 54m49s to 1h45m depending on tolerance. When using the iterative refinement process with hierarchical coresets of Algorithm 1, training times are reduced to 11m10s for the coarsest tolerance to 27m2s for the finest tolerance without any reduction in dictionary quality.

In terms of encoding performance, the processing stage has a complexity similar to the COVRA encoder [GIM12]. We measured an encoding speed ranging from 4.9 to 7.3 MVoxels/s, leading to a frame encoding time of 3.5-5.5 minutes/frame for IsoTur and MHD and 4.4-6.6 minutes/frame for Channel on a single machine.

5.2. Compression rate and distortion

In order to provide a context for the evaluation of our work, we compare our results with a fixed-rate version of the same code, as well as with a recently introduced state-of-the-art solution based on Elastic Sparse Coding (ESC) [MAG19], which is also capable of real-time performance, and represents the current state-of-the-art in compression for real-time rendering. As Marton et al. [MAG19] have already compared ESC with the major real-time and non-real-time codecs, including COVRA [GIM12]), ASTC [NLP*12], Hierarchical Vector Quantization (HVQ) [SW03], CudaCompress wavelet codec (CC) [TBR*12], ZFP [Lin14], and SZ [DC16], we do not repeat all the benchmarks here. As a reference, we include results obtained with the ZFP [Lin14] codec (V 0.5.4), which is a de-facto standard for compression of floating-point volumetric data. For ZFP, we used the fixed accuracy mode, which usually yields the best signal-to-noise ratios, and varied the absolute error tolerance (−a) to obtain the desired bit rates.

Numerical results are presented in Table 1. Compression rate

	IsoTur (4 TB)			**MHD** (4 TB)			**Channel** (23.4 TB)		
MTV-Player VAR, S_{max}=15, K=1024, M=6	**T 0.08**	**T 0.055**	**T 0.041**	**T 0.07**	**T 0.049**	**T 0.038**	**T 0.005**	**T 0.004**	**T 0.0025**
Size(GB)	70.1	91.06	111.37	70.73	92.95	111.68	416.74	487.22	649.18
bps	0.34	0.44	0.54	0.34	0.45	0.54	0.34	0.40	0.53
PSNR(dB)	67.73	69.68	70.88	52.23	54.11	55.15	57.55	58.97	60.97
MTV-Player FIX, S_{max}=15, K=1024, M=6	**S 3**	**S 4**	**S 6**	**S 3**	**S 4**	**S 6**	**S 3**	**S 4**	**S 6**
Size(GB)	69.73	85.11	115.87	69.73	85.11	111.68	408.60	498.70	678.92
bps	0.34	0.41	0.56	0.34	0.41	0.56	0.34	0.41	0.56
PSNR(dB)	63.83	65.50	67.83	48.77	50.31	52.37	53.26	54.94	57.36
ESC [MAG19] same bps, K=1024, M=8	**S 9**	**S 12**	**S 15**	**S 9**	**S 12**	**S 15**	**S 9**	**S 12**	**S 15**
Size(GB)	51.10	65.70	78.84	51.10	65.70	78.84	299.41	384.96	461.95
bps	0.35	0.45	0.54	0.35	0.45	0.54	0.35	0.45	0.54
PSNR(dB)	67.39	68.57	69.27	51.81	52.99	53.92	57.09	58.53	59.50
ESC [MAG19] same size, K=1024, M=8	**S 13**	**S 17**	**S 22**	**S 13**	**S 17**	**S 22**	**S 13**	**S 17**	**S 22**
Size(GB)	69.58	87.83	110.26	69.58	87.83	110.26	407.68	512.39	646.06
bps	0.48	0.60	0.76	0.48	0.60	0.76	0.48	0.60	0.76
PSNR (dB)	68.77	69.61	70.22	53.34	54.40	55.46	58.87	59.92	60.63
ZFP [Lin14]	**A 2.75**	**A 1.125**	**A 0.625**	**A 2.00**	**A 1.00**	**A 0.5**	**A 0.125**	**A 0.0625**	**A 0.0312**
bps	0.36	0.42	0.53	0.36	0.49	0.68	0.36	0.49	0.67
PSNR(dB)	49.24	58.92	63.36	42.67	46.65	50.92	49.24	53.53	58.04

Table 1: ***Compression rate and distortion***. *The compared codecs are MTV Player with tolerance-driven variable-rate encoding, MTV Player with fixed rate encoding at similar bits per output sample, Elastic Sparse Coding (ESC) [MAG19] with similar bits per output sample, Elastic Sparse Coding (ESC) [MAG19] with similar output file storage size, and ZFP [Lin14] at similar target bit rate. For ZFP, we used the fixed accuracy mode, which usually yields the best signal-to-noise ratios, and varied the absolute error tolerance (−a) to obtain the desired bit rates.*

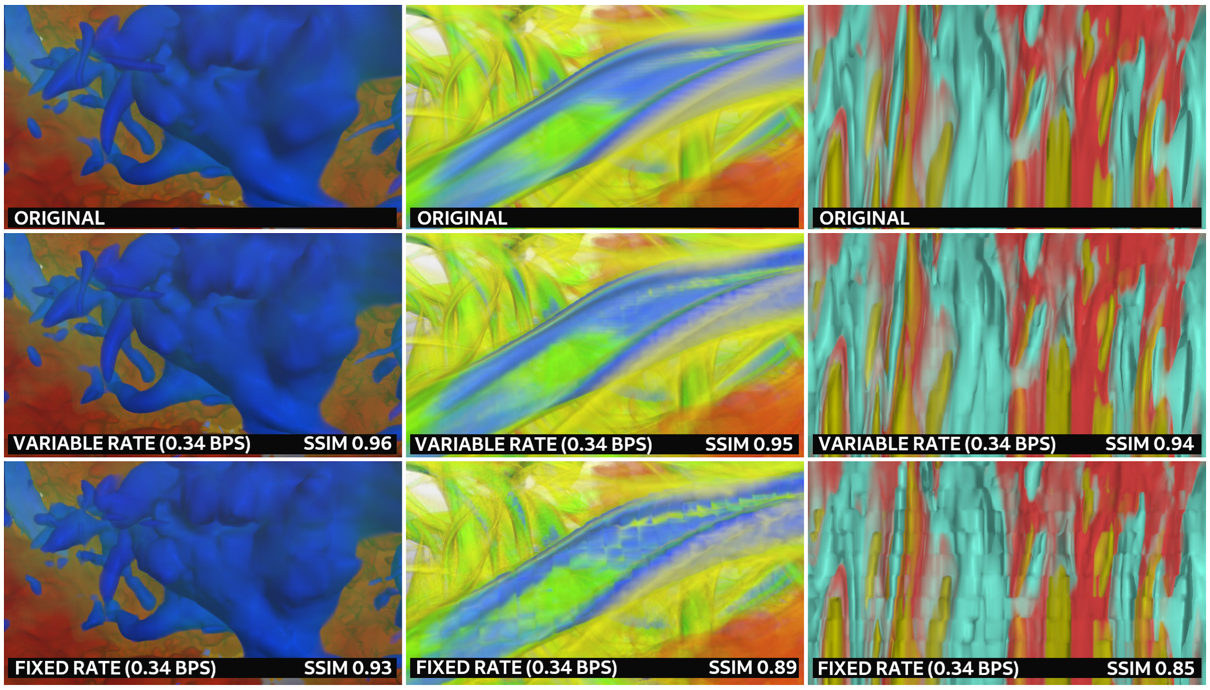

Figure 5: ***Perceptual quality assessment.*** *The SSIM values are the average of the grayscale SSIM of each color channel.*

is measured in bits per output sample (bps), while quality is measured with peak signal to noise ratio (PSNR), defined as $10\log_{10}\frac{(\max_i x_i - \min_i x_i)^2}{\frac{1}{N}\sum_i (x_i - y_i)^2}$, where x_i is the original voxel value, y_i is the approximated one and N the total number of voxels.

Each dataset has been compressed using our codec with three different tolerances using the variable-rate encoder, as well as with a fixed-rate version of the encoder, where the target sparsity has been set to the average value found by the variable-rate encoder, in order to match the achieved bit rate. Parameters for ESC and ZFP have been also set to provide a similar compression. Since the storage approach is different, for ESC, we provide two results: one in which we match the target bits/output voxel, and one in which we match the target size.

The new variable-rate codec scales well and provides considerable improvement in terms of PSNR with respect to the fixed-rate solution. The codec is also very competitive with respect to ESC. It should be noted that ESC is bitrate-driven and is thus capable of meeting hard storage constraints for each frame, while our codec

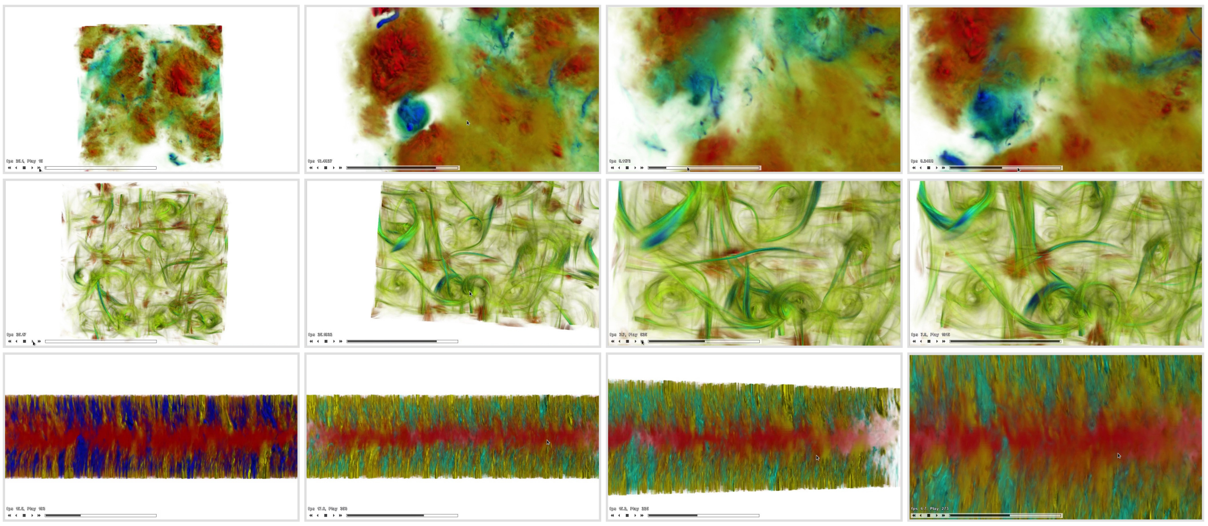

Figure 6: ***Representative frames of the accompanying video.*** *Our rendering architecture supports interactive spatial exploration, modification of the transfer function parameters, playing the sequence forwards and backwards at various speeds, and full random access to individual frames.*

is tolerance driven, and uses an adaptive bit rate, which gives more freedom but, at the same time, requires some tuning to control output size. Moreover, ESC uses non-overlapping bricks, while we replicate data at brick boundaries in order to make bricks self-contained. Our storage format is thus a bit heavier. ZFP, used here as a reference implementation, does not appear to perform that well at such extreme compression rates (close or below 0.5 bps). On the other hand, using a near-lossless setup, it could be employed in the encoder to speed-up data accesses. This is compatible with the findings of Marton et al. [MAG19].

The improved results in terms of average and maximum errors lead to improved perceptual quality during volume exploration with respect to fixed-rate solutions. Fig. 5 illustrates the visual quality obtained for our benchmark datasets, comparing fixed rate with variable rate encoding at the same compression rate of 0.34 bps. The images were taken at extreme closeups, using representative transfer functions with both transparent and opaque materials, as in the accompanying video. Note that despite the high compression rates, the essential parts, as well as details of a certain feature size can be visualized in all datasets, and that the variable-rate version is more similar to the original. As for all block-based compression techniques, artifacts appear as discontinuities among adjacent blocks, but are only clearly visible in the fixed-rate version. Deblocking techniques could be applied to further improve visual quality, at the expense of decoding and/or rendering time [MIDG14].

We also numerically assessed the visual quality provided by both encoding schemes by using the Structural Similarity metric [WBSS04], which is known to have a good correlation with perceptual quality. As in previous work [RWS09], we compute the grayscale SSIM of each color channel and use the mean as an overall distortion measure. The results obtained with the images of Fig. 5 demonstrate very good results even at such a high compression, as it has been verified that the point at which a human observer cannot determine that compression has been used hovers around an SSIM value of 95 [FWAP13]. Moreover, variable rate encoding provides better visual quality than the fixed-rate one, with SSIM values that increase by 4-9% depending on the dataset, which are considered noticeable differences.

5.3. Interactive exploration

We evaluated the rendering performance of our framework on a number of interactive inspection sequences of our three dynamic datasets, using the dataset configurations selected in Table. 1. The tests have been performed on the same desktop PC used for dictionary training, equipped with a NVIDIA GeForce 980 GTX with 4GB of video memory.

The qualitative performance of our adaptive GPU ray-caster is illustrated in the accompanying video. Because of video frame capture constraints, the sequence is recorded using a window size of 1024×576 pixels. In all recorded sequences, we used a 0.5 voxel/pixel accuracy. Representative frames are in Fig. 1 and Fig. 6. As shown in the video, the system is fully interactive. It is possible to translate, rotate, and scale the volumes, to change the transfer function while playing back animations at various speeds, moving back-and-forth in time, and jumping at different time-steps. Frame rates are generally well above 20Hz, varying from 8Hz for medium-range closeups where most of the full dataset is visible at high resolution to above 26Hz for overall views or extreme closeup views of models, where we can better exploit level-of-detail and view culling to reduce upload, decode, and render overhead. Data loading is performed in parallel with the decoding and rendering kernels. GPU profiling reveals an occupancy of 41% for decoding and 59% for rendering a single frame. As we do not exploit temporal coherence, performance does not change depending on whether animations are played backwards and forwards, or when they are rendered at higher speed. Moreover, thanks to the lack of incremental updates, no dynamic artifacts (e.g., partial refinements) are visible in the animation. Such a streaming architecture is, however, limited

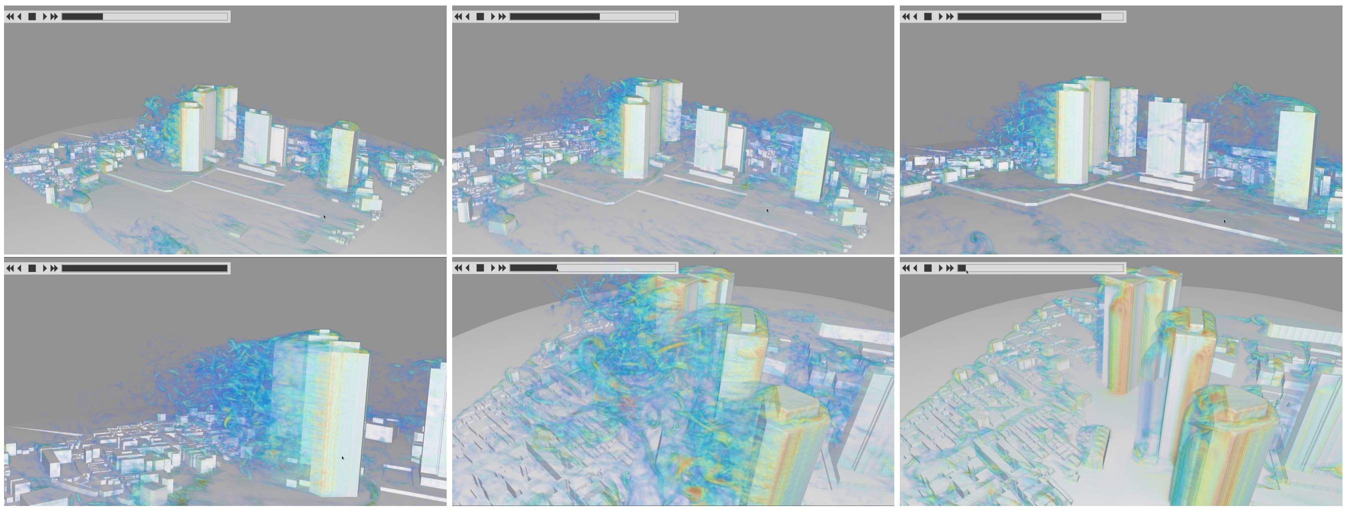

Figure 7: ***Example of application to urban CFD simulation.*** *Frames from an interactive sequence of spatio-temporal exploration of the vorticity field around buildings at a urban scale.*

by the amount of data that is streamed, decoded, and rendered on a per-frame basis. See accompanying video for more details.

6. Conclusions

We have presented a novel approach for supporting fully interactive exploration with non-trivial temporal access of massive time-varying rectilinear scalar volumes on commodity platforms. Instead of looking at maximum compression and adaptivity, we streamline the rendering loop, by using a time-independent codec that produces a highly compressed format and asynchronously sending to GPU the required frame portions in few batches, to be decoded and rendered at interactive speeds. Our variable-rate encoding scheme based on sparse coding volume blocks using a learned dictionary deals with bandwidth and memory limitations, while providing competitive perceptual and signal reconstruction quality at similar compression ratios with respect to current state-of-the-art fixed-rate or constrained variable-rate solutions supporting real-time performance. Furthermore, thanks to the scalability at all the stages of the pipeline, the presented architecture is capable of processing and visualizing massive datasets, as demonstrated by our results on terascale turbulence simulations, freely explored by rapidly moving in time, space and transfer function parameters.

Our current work is aimed at using the approach for the exploration of simulation data, especially in the area of large scale CFD simulation. Fig. 7 and the accompanying video illustrate our preliminary results, which show the possibility of exploring in real-time the vorticity field around buildings at a urban scale.

Acknowledgments. The authors would like to warmly thank Peter Lindstrom for making available his excellent ZFP code. Datasets are courtesy of the Johns Hopkins Turbulence Database (JHTDB) initiative. The urban dataset is courtesy of the Architectural Institute of Japan. We also acknowledge the contribution of Sardinian Regional Authorities under projets VIGECLAB and TDM (POR FESR 2014-2020 Action 1.2.2).

References

[AEB06] Aharon M., Elad M., Bruckstein A.: K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE TSP 54*, 11 (2006), 4311–4322. 2, 3

[AFdMSP18] Amorim P., Franco de Moraes T., Silva J., Pedrini H.: *Out-of-Core Rendering of Large Volumetric Data Sets at Multiple Levels of Detail: Applications and Computational Techniques*. Springer, 01 2018, pp. 191–215. 2

[BHP15] Beyer J., Hadwiger M., Pfister H.: State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37. 2

[BRGIG*14] Balsa Rodriguez 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. 2

[BRLP18] Ballester-Ripoll R., Lindstrom P., Pajarola R.: TTHRESH: Tensor compression for multidimensional visual data. *arXiv preprint arXiv:1806.05952* (2018). 2

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. I3D* (2009), pp. 15–22. 2, 5

[Cra] Craighead M.: Gl_nv_texture_compression_vtc. OpenGL Extension Registry. 2

[CWW11] Cao Y., Wu G., Wang H.: A smart compression scheme for GPU-accelerated volume rendering of time-varying data. In *Proc. IEEE ICVRV* (2011), pp. 205–210. 3

[DC16] Di S., Cappello F.: Fast error-bounded lossy HPC data compression with SZ. In *Proc. IEEE IPDPS* (2016), pp. 730–739. 6

[Eng11] Engel K.: CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *Proc. IEEE LDAV* (2011), pp. 123–124. 2

[FM07] Fout N., Ma K.-L.: Transform coding for hardware-accelerated volume rendering. *IEEE TVCG 13*, 6 (2007), 1600–1607. 2, 3

[FSK13] Fogal T., Schiewe A., Kruger J.: An analysis of scalable GPU-based ray-guided volume rendering. In *Proc. IEEE LDAV* (Oct 2013), pp. 43–51. 2

[FWAP13] Flynn J., Ward S., Abich Julian I., Poole D.: Image quality assessment using the SSIM and the just noticeable difference paradigm. In *Engineering Psychology and Cognitive Ergonomics. Understanding Human Cognition*, vol. 8019 of *LNCS*. Springer, 2013, pp. 23–30. 8

[GG16] GUTHE S., GOESELE M.: Variable length coding for GPU-based direct volume rendering. In *Proc. VMV* (2016), pp. 77–84. 2

[GIM12] GOBBETTI E., IGLESIAS GUITIÁN J., MARTON F.: COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum 31*, 3/4 (2012), 1315–1324. 2, 3, 4, 5, 6

[GS01] GUTHE S., STRASSER W.: Real-time decompression and visualization of animated volume data. In *Proc. IEEE Vis* (2001), IEEE, pp. 349–572. 2

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE TVCG 18*, 12 (2012), 2285–2294. 2

[IGM10] IGLESIAS GUITIÁN J. A., GOBBETTI E., MARTON F.: View-dependent exploration of massive volumetric models on large scale light field displays. *The Visual Computer 26*, 6–8 (2010), 1037–1047. 2

[Iri06] IRION R.: The terascale supernova initiative: Modeling the first instance of a starâĂŹs death. *SciDAC Review 2*, 1 (2006), 26–37. 1

[JEG12] JANG Y., EBERT D. S., GAITHER K. P.: Time-varying data visualization using functional representations. *IEEE TVCG 18*, 3 (2012), 421–433. 3

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. Graphics Hardware* (2002), pp. 7–15. 2

[KLW*08] KO C.-L., LIAO H.-S., WANG T.-P., FU K.-W., LIN C.-Y., CHUANG J.-H.: Multi-resolution volume rendering of large time-varying data using video-based compression. In *Proc. IEEE Pacific Vis* (2008), pp. 135–142. 3

[Lin14] LINDSTROM: Fixed-rate compressed floating point arrays. *IEEE TVCG 20*, 12 (2014), 2674–2683. 2, 6, 7

[LMC02] LUM E. B., MA K.-L., CLYNE J.: A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE TVCG*, 3 (2002), 286–301. 2

[LPW*08] LI Y., PERLMAN E., WAN M., YANG Y., MENEVEAU C., BURNS R., CHEN S., SZALAY A., EYINK G.: A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, 9 (2008). 1, 6

[MAG19] MARTON F., AGUS M., GOBBETTI E.: A framework for gpu-accelerated exploration of massive time-varying rectilinear scalar volumes. *Computer Graphics Forum 38*, 3 (2019). 2, 3, 4, 6, 7, 8

[MIDG14] MARTON F., IGLESIAS GUITIÁN J., DIAZ J., GOBBETTI E.: Real-time deblocked GPU rendering of compressed volumes. In *Proc. VMV* (2014), pp. 167–174. 8

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K.: A GPU-supported lossless compression scheme for rendering time-varying volume data. In *Proc. Volume Graphics* (2010), pp. 109–116. 3

[MS00] MA K.-L., SHEN H.-W.: Compression and accelerated rendering of time-varying volume data. In *Proc. International Workshop on Computer Graphics and Virtual Reality* (2000), pp. 82–89. 3

[NIH08] NAGAYASU D., INO F., HAGIHARA K.: Two-stage compression for fast volume rendering of time-varying scalar data. In *Proc. GRAPHITE* (2008), pp. 275–284. 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *Proc. HPG* (2012), pp. 105–114. 6

[PGEK17] PARK J., GUTENKO I., E. KAUFMAN A.: Transfer function-guided saliency-aware compression for transmitting volumetric data. *IEEE Transactions on Multimedia PP* (09 2017), 1–1. 2

[PK09] PARYS R., KNITTEL G.: Giga-voxel rendering from compressed data on a display wall. In *Proc. WSCG* (2009). 2

[PLK*18] PULIDO J., LIVESCU D., KANOV K., BURNS R. C., CANADA C., AHRENS J. P., HAMANN B.: Remote visual analysis of large turbulence databases at multiple scales. *J. Parallel Distrib. Comput. 120* (2018), 115–126. 3

[RTW13] REICHL F., TREIB M., WESTERMANN R.: Visualization of big SPH simulations via compressed octree grids. In *Proc. IEEE Big Data* (2013), pp. 71–78. 2

[RWS09] RAJASHEKAR U., WANG Z., SIMONCELLI E. P.: Quantifying color image distortions based on adaptive spatio-chromatic signal decompositions. In *Proc. IEEE ICIP* (2009), pp. 2213–2216. 8

[RZE08] RUBINSTEIN R., ZIBULEVSKY M., ELAD M.: *Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit*. Tech. rep., CS Technion, 2008. 4

[SBN11] SHE B., BOULANGER P., NOGA M.: Real-time rendering of temporal volumetric data on a GPU. In *Proc. IEEE InfoVis* (2011), pp. 622–631. 1, 3

[She06] SHEN H.-W.: Visualization of large scale time-varying scientific data. *Journal of Physics 46*, 1 (2006), 535–544. 3

[SIM*11] SUTER S., IGLESIAS GUITIÁN J., MARTON F., AGUS M., ELSENER A., ZOLLIKOFER C., GOPI M., GOBBETTI E., PAJAROLA R.: Interactive multiscale tensor reconstruction for multiresolution volume visualization. *IEEE TVCG 17*, 12 (2011), 2135–2143. 2

[SJ94] SHEN H.-W., JOHNSON C. R.: Differential volume rendering: A fast volume visualization technique for flow animation. In *Proc. IEEE Vis* (1994), pp. 180–187. 2

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proc. IEEE Vis.* (2003), pp. 293–300. 2, 6

[TBR*12] TREIB M., BURGER K., REICHL F., MENEVEAU C., SZALAY A., WESTERMANN R.: Turbulence visualization at the terascale on desktop PCs. *IEEE TVCG 18*, 12 (2012), 2169–2177. 2, 3, 6

[WBSS04] WANG Z., BOVIK A., SHEIKH H., SIMONCELLI E.: Image quality assessment: from error visibility to structural similarity. *IEEE TIP 13*, 4 (2004), 600 –612. 8

[Wes95] WESTERMANN R.: Compression domain rendering of time-resolved volume data. In *Proc.IEEE Vis* (1995), pp. 168–175. 3

[WF08] WEISS K., FLORIANI L.: Modeling and visualization approaches for time-varying volumetric data. In *Proc. Advances in Visual Computing* (2008), pp. 1000–1010. 1, 2

[WGLS05] WANG C., GAO J., LI L., SHEN H.-W.: A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Proc. Volume Graphics* (2005), pp. 11–19. 3

[WWS03] WOODRING J., WANG C., SHEN H.-W.: High dimensional direct rendering of time-varying volumetric data. In *Proc. IEEE Vis* (2003), pp. 417–424. 2

[WWS*05] WANG H., WU Q., SHI L., YU Y., AHUJA N.: Out-of-core tensor approximation of multi-dimensional matrices of visual data. *ACM TOG 24*, 3 (July 2005), 527–535. 2

[WYM08] WANG C., YU H., MA K.-L.: Importance-driven time-varying data visualization. *IEEE TVCG 14*, 6 (2008), 1547–1554. 3

[WYM10] WANG C., YU H., MA K.-L.: Application-driven compression for visualizing large-scale time-varying data. *IEEE CGA 30*, 1 (2010), 59–69. 3

[YNV08] YELA H., NAVAZO I., VAZQUEZ P.: S3Dc: A 3Dc-based volume compression algorithm. *Computer Graphics Forum* (2008), 95–104. 2

[YZW*17] YU S., ZHANG S., WANG K., XIA Y., ZHANG H.: An efficient and fast GPU-based algorithm for visualizing large volume of 4D data from virtual heart simulations. *Biomedical Signal Processing and Control 35* (2017), 8–18. 2