Computers & Graphics (2020)

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Interactive Spatio-Temporal Exploration of Massive Time-Varying Rectilinear Scalar Volumes based on a Variable Bit-Rate Sparse Representation over Learned Dictionaries

Jose Díaz[a,*], Fabio Marton[b,**], Enrico Gobbetti[b,**]

[a]*Digital Care Research Group, UVic-UCC, Spain*
[b]*Visual Computing Group, Center for Advanced Studies, Research and Development in Sardinia (CRS4), Cagliari, Italy*

ARTICLE INFO

Article history:
Received 13 December 2019
Received in final form 2 March 2020
Accepted 4 March 2020

Keywords: Direct volume rendering, Time-varying data, Compression, Sparse coding, Learned dictionary

ABSTRACT

We introduce a novel approach for supporting fully interactive non-linear spatio-temporal exploration of massive time-varying rectilinear scalar volumes on commodity platforms. To do this, we decompose each frame into an octree of overlapping bricks. Each brick is further subdivided into smaller non-overlapping blocks compactly approximated by quantized variable-length sparse linear combinations of prototype blocks stored in a learned data-dependent dictionary. An efficient tolerance-driven learning and approximation process, capable of computing the tolerance required to achieve a given frame size, exploits coresets and an incremental dictionary refinement strategy to cope with datasets made of thousands of multi-gigavoxel frames. The compressed representation of each frame is stored in a GPU-friendly format that supports direct adaptive streaming to the GPU with spatial and temporal random access, view-frustum and transfer-function culling, and transient and local decompression interleaved with ray-casting. Our variable-rate codec provides high-quality approximations at very low bit-rates, while offering real-time decoding performance. Thus, the bandwidth provided by current commodity PCs proves sufficient to fully stream and render a working set of one gigavoxel per frame without relying on partial updates, thus avoiding any unwanted dynamic effects introduced by current incremental loading approaches. The quality and performance of our approach is demonstrated on massive time-varying datasets at the terascale.

1. Introduction

Interactive visual exploration of very large time-varying datasets is crucial to understand scientific simulation results [1, 2]. Such data, commonly represented as a sequence of time-varying rectilinear scalar volumes, normally present thousands of time steps and billions of voxels per frame [3, 4]. Meeting interactivity constraints when rendering such massive datasets is very hard, especially when showing the animated sequence. 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 variable-rate approach of Marton et al. [5], 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).

*Corresponding author: Mail: jose.diaz@uvic.cat
**Corresponding authors: Mail: enrico.gobbetti@crs4.it; fabio.marton@crs4.it; WWW: www.crs4.it/vic/

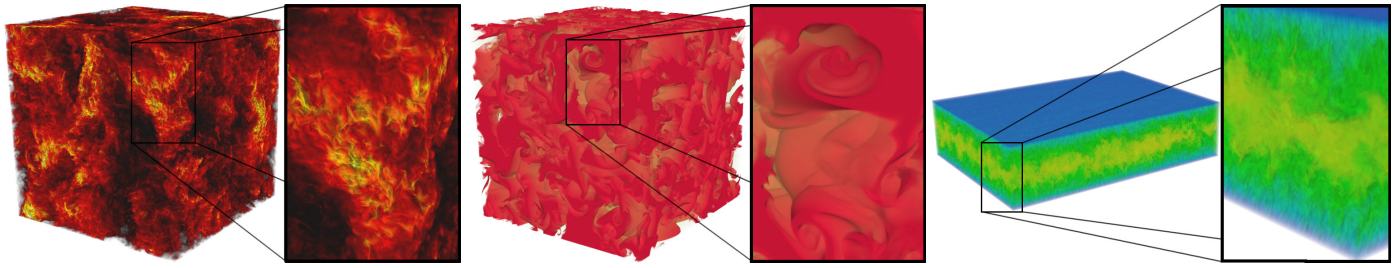

Fig. 1. ***Real-time exploration.*** **Our multiresolution compression-domain GPU volume rendering architecture supports interactive random-access exploration of massive time-varying rectilinear scalar volumes on commodity platforms. From left to right: velocity magnitude field of forced isotropic turbulence simulation (*ISO*, 1024 time steps 1024^3, float, 4TB) compressed to 93.6 GB (0.45 bps, PSNR 46.19 dB for frame 256); density field of a homogeneous buoyancy-driven turbulence simulation (*HBDT*, 1010 time steps 1024^3, float, 4TB) compressed to 92.7 GB (0.45 bps, PSNR 41.48 dB for frame 256); $(2048 \times 512 \times 1536) \times 4000$ Channel Flow simulation (23.4 TB) compressed to 551.7 GB (0.45 bps, PSNR 49.78 dB for frame 256). The main images show an overall view of the full dataset, while the inset provides an illustration of the amount of detail available for exploration.**

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 (in the rest of the paper, our approach is referred as *MTV-Player* for *Massive Time-varying Volume Player*).

By encoding frames in an I/O and GPU-friendly compressed format that supports 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 bricked octree. The bricks, used as I/O, culling, and rendering units, have a 2-voxel apron (i.e., a border around each brick that duplicates the values across brick boundaries) in order to allow exploiting texturing operations for trilinear filtering and gradient computations. Each brick is further subdivided into smaller non-overlapping blocks, which are compactly approximated by a quantized variable-length sparse linear combinations of prototype blocks stored in a data-dependent dictionary learned from the input sequence (Sec. 3). In order to meet bandwidth constraints, we estimate per-frame compression tolerances from the required frame size. 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 encoding scheme can support scalable dictionary learning on massive datasets and meet size constraints through automatic tolerance computation. 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 our codec can be integrated in an out-of-core real-time rendering architecture, capable of fully streaming and rendering dynamic representations of gigavoxel-sized frames without relying on partial updates of individual frames. This article is an invited extended version of our STAG 2019 contribution [6]. We here provide a more thorough exposition, but also significant new material, including an efficient incremental dictionary learning method for time-varying sequences, an algorithm to estimate tolerance from size constraints, and an improved quantized encoding of sparse-coded blocks. Furthermore, the evaluation has been significantly extended.

The major limitations of our 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 [1], compression-based direct volume rendering [7] and GPU-based large-scale volume visualization [8].

2.1. Data compression for GPU-accelerated DVR

State-of-the-art solutions are based on compressing and storing data in multiresolution out-of-core structures such as octrees [9, 10, 11, 12], bricks [13], hierarchical grids of bricks [14, 15] or hierarchical tiled 3D grids [5], 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 based on some form of block truncation coding (BTC [16]) and supporting random access and interpolation [17, 18, 19], at the cost of limiting achievable compression and rate-distortion performance [20, 21].

Vector quantization solutions [22, 23, 24, 25] also support real-time visualization, since decoding can be achieved by simple dictionary lookups that can be accelerated by texture caches. Their quality is, however, limited by the dictionary size [26]. Guthe and Goesele [24] combined vector quantization with lossless compression to support larger dictionaries with respect to uncompressed vector quantization at the same bit-rate. While good results are achieved with 8- or 12-bit homogeneous datasets (e.g., CT scans), vector quantization limitations still apply, and severe blocking artifacts appear on floating point simulation data at low bit-rates. Moreover, compression times, in the range of many hours per gigavoxel, are not compatible with time-varying datasets at the terascale.

Several more advanced codecs, in particular the ones based on Wavelets [27, 28] and tensor approximation solutions [29, 30] achieve excellent performance especially when combined with advanced entropy coding methods. However, at low bit rates and due to their parallel decoding complexity, even the fastest methods [13] are far from being able to sustain real-time streaming of large dynamic volumes on current hardware [5]. For this reason, interactive systems are typically forced to amortize decompression over multiple frames, which is not a suitable solution for time-varying data [5]. In order to distribute coding efforts into areas deemed more important, Park et al. [31] have proposed to improve performance with a saliency-aware codec, 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 their effectiveness to achieve state-of-the-art compression while supporting real-time decoding, as demonstrated by the COVRA architecture [11]. Since the dictionary is learned from the input data, the number of terms needed to obtain a good approximation at low bit rates is much lower than the ones needed 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. However, COVRA employs a fixed-rate approach, which is sub-optimal for datasets that exhibit wide spatial variations in data complexity, a common feature in many simulations. Tensor decomposition is also learned from data, but it imposes limitations on decoding locality and achieves a much lower decoding speed [29, 30]. Recently, Marton et al. [5] have proposed a variation that improves COVRA's fixed-rate scheme through a constrained variable-rate encoder. This approach adapts bit rates within fixed-size pages and provides a better bit allocation scheme. In our original contribution [6], we have shown 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 distribution with respect to page-constrained schemes. However, selecting the tolerance required to meet the size constraints dictated by the need to stream large frames at interactive speed proves very hard. For this reason, we introduce here a coreset-based approach to efficiently estimate a per-frame tolerance. Moreover, we refine the quantized representation to further improve rate-distortion performance.

2.2. Time-varying data exploration

Visualizing time-varying datasets has usually been addressed with temporal-coherent compression techniques [32, 33, 34, 35, 36], 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 difficults random access to specific time-steps. For this reason, 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 [37, 38, 39, 40, 41, 42, 43, 2, 44], but restrictions on the maximum size of data are still present, even when combining both approaches [20, 45, 46, 47].

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 [11, 13, 48, 5]. With no temporal dependency, full random access to single time-steps is guaranteed. As in the COVRA architecture [11], our technique encodes volume blocks using a sparse representation based on a dictionary learned by means of the K-SVD algorithm [26]. However, instead of using a fixed-rate compression scheme, our variable-rate encoding provides a better trade-off 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 [13], which includes run-length and entropy encoding, or the recent method by Pulido et al. [48] 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. [5] also supports high-quality exploration of massive time-varying datasets in both desktop and mobile devices. In this case, they use a variable-rate encoding scheme based on sparse coding with fixed-size pages and a hierarchy of grids. This feature enables 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. Since the drawback of using sparse coding from learned dictionaries on massive datasets is the lengthy dictionary learning time, we have recently proposed a faster training scheme based on a hierarchy of coresets [6], which is however limited to have a single dictionary for the entire sequence. In this work, we significantly improve this approach for time-varying data by employing a different dictionary per frame, using an efficient incremental dictionary learning method, which quickly adapts the dictionary of a given frame starting from the dictionary of the previous one. By applying this scheme to small frame batches, parallel dictionary construction is guaranteed.

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. Each time step is encoded into a bricked octree, compressed and stored in 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 up to a prescribed error tolerance by sparse linear combinations of prototype blocks stored in a per-frame data-dependent dictionary $\boldsymbol{D}$. 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_{train}, which bounds the amount of memory used for training, the dictionary size $K \geq M^3$, and a threshold $\epsilon \geq 0$ to bound the reconstruction error of each block. Since determining ϵ to meet specific size constraints may be cumbersome, we introduce a method based on coresets to automatically determine it given a pre-determined per-frame size.

3.1. Dictionary learning background

In our approach, an over-complete dictionary $\boldsymbol{D}$ 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 $\boldsymbol{y} \in \mathbb{R}^m$ 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:

$$\min_{\boldsymbol{D},\boldsymbol{\lambda}_i} \sum_i w_i \, \|\boldsymbol{y}_i - \boldsymbol{D}\boldsymbol{\lambda}_i\|_2^2 \tag{1}$$

subject to:

$$\forall i, \|\boldsymbol{\lambda}_i\|_0 \text{ is the minimum such that } \|\boldsymbol{y}_i - \boldsymbol{D}\boldsymbol{\lambda}_i\|_2^2 \leq \epsilon^2 \tag{2}$$

where w_i represents the weight of each training sample and $\|\boldsymbol{\lambda}_i\|_0$ is the number of non-zero entries of $\boldsymbol{\lambda}_i$. A variation of the K-SVD algorithm [26] is used to train the dictionary $\boldsymbol{D}$, where each prototype block is mapped to a unitary column vector $\boldsymbol{d}_i \in \mathbb{R}^m$ and maintained at zero mean. For sparse coding blocks given the learned dictionary, we employ the batch-OMP algorithm [49], which supports encoding with a prescribed tolerance. In the following sections, we introduce techniques to efficiently compute an adapted per-frame dictionary at a small cost, as well as a scalable method to determine the tolerance required to achieve the best results within a given per-frame size budget.

3.2. Dictionary learning for frame sequences

Compressing a dataset requires the computation of a sparsifying dictionary D, which is efficient if closely modeling the data. Learning this dictionary is however costly. Similarly to previous work [11, 5], in order to reduce the complexity of dictionary learning, we perform it on a weighted subset of the original samples (i.e., a coreset) instead of on all the input samples. Since constant blocks can be trivially encoded due to the fact that we remove the average, we estimate for each input block $\boldsymbol{b}_i$ the potential residual error $e_i = \|\boldsymbol{b}_i - \bar{y}\boldsymbol{1}\|_2^2$, and in order to build a coreset of size C_{train}, we pick training samples with a probability proportional to e_i using a one-pass streaming method based on weighted reservoir sampling [50], assigning a weight proportional to the reciprocal of the picking probability to account for the non-uniform sampling.

The K-SVD method employed for dictionary learning, then, alternatively iterates between two sub-problems solved by heuristic greedy methods over the coreset samples: sparse coding, which finds the best $\boldsymbol{\lambda}_i$ given a fixed dictionary, and dictionary updating, which updates one column of $\boldsymbol{D}$ at a time in order to minimize the representation error given the current sparse representation. For sparse coding, we employ the same batch-OMP algorithm [49] used for final encoding, while for dictionary updating we employ the Approximate K-SVD algorithm [49], modified to take into account training sample weights, as in previous work [11, 5, 6]. Table 1 shows the time required for dictionary learning using 100 K-SVD iterations, for different coreset sizes, and for the frame depicted in Fig 1 of the three test datasets in a typical configuration (block size $M = 6$, brick size of $B = 32$ voxels, and dictionary size $K = 2048$). Dictionary learning time, measured on the machine described in Sec. 5.2, is close to linear in coreset size, and very good dictionaries are obtained with extremely small coresets, since PSNR (Peak Signal-to-Noise-Ratio) reduction saturates very quickly. For this paper, we have thus used coresets of 16MVoxels, which are far below 2% of the frame size for all datasets.

	ISO			HBDT			CHAN		
Coreset Size	%	Time	PSNR	%	Time	PSNR	%	Time	PSNR
2	0.20%	65	44.83	0.20%	63	37.78	0.13%	64	47.12
4	0.39%	72	45.25	0.39%	71	38.24	0.26%	75	47.51
8	0.78%	87	45.49	0.78%	89	38.51	0.52%	89	47.77
16	1.56%	127	45.65	1.56%	175	38.71	1.04%	129	47.93
32	3.13%	206	45.76	3.13%	213	38.83	2.08%	206	48.03

Table 1. ***Achieved PSNR as a function of the coreset size*. Data is reported for a variable-rate encoding at 0.45bps.**

However, even with such a large reduction, dictionary learning times on such small coresets are still non-negligible for single frames and on-par or superior to the encoding step for all blocks contained in an entire frame [5]. For this reason, Díaz et al. [6] proposed to only learn a single dictionary for the entire sequence, using a coreset extracted from multiple frames. This approach, however, is valid only in particular cases, where little variations

occurs during simulation, losing the advantage of learned dictionaries in cases where significant qualitative evolution exist. For instance, in the HBDT simulation (see Fig. 1 and Sec. 5), two fluids are initialized as random homogeneous blobs, and turbulence is generated as the two fluids start moving in opposite directions due to differential buoyancy forces, until when the fluids become molecularly mixed, and buoyancy forces decrease, decaying also the turbulence. Thus, the characteristics of all fields and of related optimal dictionaries vary widely across the simulation.

In this work, we propose to exploit the incremental nature of the K-SVD algorithm to drastically reduce learning time for general simulation results. In our approach, we decompose the input sequence in small batches of consecutive frames (32 in this paper), which are then processed sequentially. The first frame of the batch (key frame), learns the associated dictionary using a frame-specific coreset of size C_{train}, starting from an initial random dictionary and 100 K-SVD iterations, which are sufficient to arrive to convergence. Instead, the subsequent frames in the batch start from the dictionary learned in the previous frame and, using again a frame-specific coreset of size C_{train}, refines the dictionary for just 5 iterations. This approach is motivated by the fact that it is expected that even widely varying simulations are smoothly evolving, and we can also expect that one frame's optimal dictionary would be not far from the optimal dictionary of the next frame. By applying this sequential approach to small frame batches, we support parallelization, as all batches can be constructed in parallel, and we remove problems due to drift. The validity of this approach is proved by our experimental results. In particular, the average learning cost for a batch of 32 frames drops from 4000s (120s/frame) for the 3 datasets to 318s (10s/frame), while the achieved PSNR when using the incrementally learned dictionary is within ±0.1% of the PSNR achieved when independently learning all frames. Moreover, for significantly varying datasets, such as HBDT, having a per-frame dictionary is measurably better than using a single dictionary for multiple frames. For instance, the PSNR at frame 32 of the HBDT sequence is a full 1% lower when using a single dictionary for the first 32 frames than when using a per-frame dictionary. Uniform selection of keyframes to subdivide learning into batches is, thus, a simple but very effective solution: it guarantees load balancing, does not require a prior analysis of the dataset, and achieves a quality almost indistinguishable from full processing at a fraction of the cost.

3.3. Determining tolerance given size constraints

The tolerance-driven approach used in this paper allows for ensuring a constant reconstruction quality for each frame. However, in order to be usable within bandwidth-critical operations, which occur when transferring the dataset from a remote server to a local renderer, or when transferring data from storage to graphics memory at each frame during dynamic operations, it is essential to meet storage size constraints. For instance, a single 1 Gvox frame (or working set) would require 64 MB at 0.5 bits per sample (bps), translating to 64GB for streaming a 1K-frames animation from a non-local server. Moreover, supporting a 10 frames/s animation requires a codec able to load, decompress

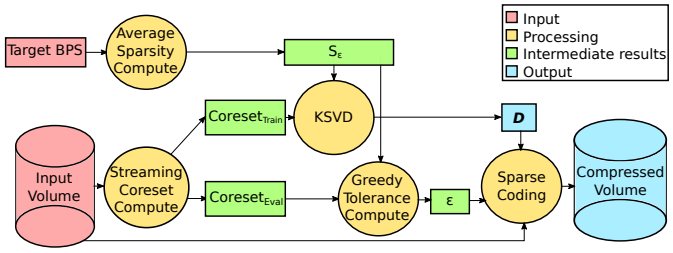

Fig. 2. ***Compression pipeline.*** **Variable-rate compression is achieved by efficiently computing target tolerance from a target size constraint using coresets.**

and render at least 10 Gvox/s. For this reason, previous works have favored either fixed-rate compression [11], or variable-rate compression within fixed-rate pages [5]. Variable-rate encoding has been also employed [6], but achieving compliance with size constraints required manual tuning, difficult for single frames (and far from optimal for large sequences). Instead, in this paper we use a method for computing the target tolerance from a target size constraint.

The algorithm (Fig. 2) first translates the desired bit-rate into a target floating point average sparsity S_ϵ based on the knowledge of the encoding (Sec. 3.4), and then learns the dictionary $\boldsymbol{D}$ using a fixed target sparsity of round(S_ϵ). At negligible cost, during the same streaming pass that extracts the coreset for training, we also extract a second coreset of size C_{eval}, with a uniform picking probability. After learning the dictionary $\boldsymbol{D}$, we estimate, with a greedy algorithm, the tolerance required for encoding the frame at a specified target bit-rate. For each of the elements in the coreset, we compute the initial solution, using zero sparsity, and insert it into a priority queue sorted by maximum residual error. We then proceed by iteratively removing the top candidate from the queue, executing one step of the Batch-OMP method to compute the next solution to push into the queue. The method stops when the overall size constraint is met, i.e., the average sparsity is greater or equal to S_ϵ. The final tolerance ϵ for variable-rate encoding is then set to the largest residual of the computed representation. We have experimentally determined that a coreset of $8Mvox$ is sufficient to always determine a tolerance that leads to a compressed representation within 1% of the target size, spending only $4s/frame$.

3.4. Dataset encoding

Given a sparsifying dictionary, the compressed representation is achieved by sparse coding all blocks and compactly coding the dictionary and the block representation. It should be noted that, in our case, we should strive at maximum decoding performance, rather than maximum compression, because of the need to support real-time decoding during rendering. In particular, we do not apply any bit transformation and entropy coding.

The dictionary is very small with respect to the frame and does not require particular compression. In our current code, we store dictionaries in formats directly supported by the GPU (floating point or 16-bit quantization), with no form of delta encoding. In the settings used for the paper, the dictionary accounts for 800KB-1600KB/frame, i.e. just a few percent of our typical compressed frame size. Introducing more elaborate

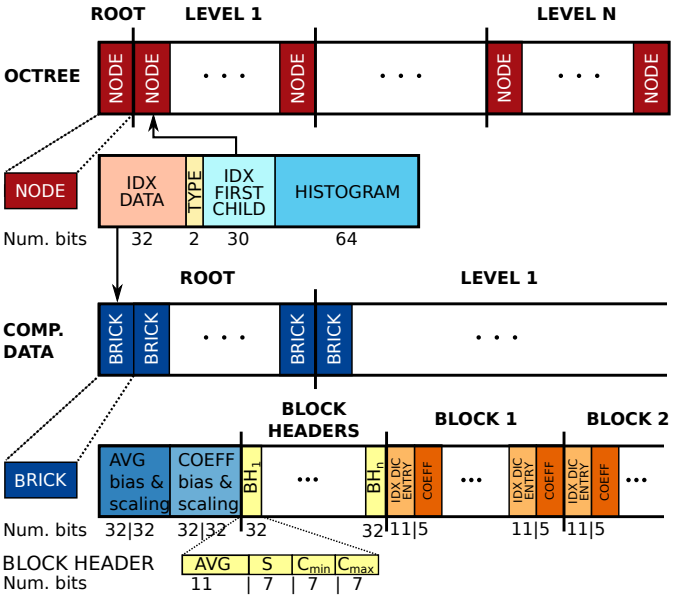

Fig. 3. ***Encoded format.*** **Each octree is stored in breadth-first order (and Morton order inside each level) to support efficient partial GPU uploading up to a run-time-determined levels. The first data segment stores the octree-structure using 128 bits per non-empty node, while the second data segment stores a variable-bit-rate representation of each non-empty brick, composed of a header plus a quantized sparse representation of its blocks. Each block has a 32 bits header containing average, sparsity, coefficient min and max bits.**

compression (e.g., multi-frame encoding, or lossy compression) would achieve a little gain. Note that, when using 16-bit format, the dictionary is quantized before encoding the frames.

The off-line storage of each octree of bricks (see Fig. 3) is designed to be compact, GPU-decodable, and traversable so that potentially visible portion of the dataset 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 the bricks corresponding to some specific level-of-detail. A 32-bit word-bounded coding scheme is employed to simplify parallel decoding on a GPU (Sec. 4).

The data storage for non-empty bricks contains the sparse representation of the contained blocks. Compression is achieved by storing for each block only the index and value of the non-empty coefficients $\boldsymbol{\lambda}_i$ and by quantizing their values, as well as the average value of the block $\bar{y}$. Thus, each brick encoding 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 dequantizing the coefficients $\boldsymbol{\lambda}_i$ of all blocks. The dequantization header is followed by two data segments. In the first segment, which contains one 32 bits entry per encoded block, we encode four values for each block: the sparsity s_i, the quantized average and the remaining bits are used for better encoding the min and max values of its coefficients. The sparsity (i.e., the number of non-zero coefficients) uses 7 bits/block for a maximum sparsity of $S_{max} = 127$, while the average uses 11 bits/block. In the second segment we encode each block as a sequence s_i (16-bit codes), with the upper bits (11 for a 2048-entry dictionary) providing the index of the prototype block in the dictionary, and the lower bits for the coefficient value. Pairs of values and indexes can be read using a single 32-bit word load operation. Scalar quantization is applied by combining information at brick, block, and individual coefficient level, as explained in Sec. 4.2.

3.5. Parallel construction

The proposed method can be efficiently parallelized on current multi-core machines and clusters. In our implementation, we assume that the original uncompressed data is stored as a separate volume per frame, and that compression is done on one or more multi-core machines. The time sequence is partitioned into small batches of 32 consecutive frames. A coordinator process distributes these batches to a number of workers, which perform dictionary learning, sparse coding, and final encoding and storage. Load balancing is ensured if the coordinator initially seeds each worker with a batch and subsequently assigns the next batch to the worker that signal completion. Each worker sequentially computes the representation of each assigned frame in order to exploit the incremental dictionary computation. However, the encoding process is massively parallel and requires minimum memory, as it can be performed in parallel for all the bricks that compose each octree, fully exploiting the multi-core architecture. Coordinator and workers may reside on the same multi-core machines or on a small cluster, since all workers are independent and very little communication is required between workers and coordinator.

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 (eventually moving it locally from a server before rendering), and we exploit efficient disk-to-GPU batched data transfer to quickly and freely achieve random access and non-trivial backward/forward/accelerated temporal exploration of time-varying sequences.

Algorithm 1 *Process to render a single time-step*

Input: Time t, view V, projection P, transfer function TF
Output: Updated frame buffer
clear_frame_buffer()
$D_t \leftarrow$ Dictionary of frame t
async_host_to_device_memcpy(D_t)
$O_t \leftarrow$ Octree of frame t
$\mathcal{F}_t$ Forest of subtrees of bounded size of O_t
for each octree $o_i \in \mathcal{F}_t$ in front-to-back-order **do**
  if is_visible(o_i, V, P, TF) **then**
    (idx, $data\_ranges$) $\leftarrow$ adaptive_selection(o_i, V, P, TF)
    (idx, $data\_ranges$) $\leftarrow$ merge_batches(idx,$data\_ranges$)
    async_host_to_device_memcpy(idx)
    async_host_to_device_memcpy($data$[$data\_ranges$])
    $dec\_bricks\_tex3D \leftarrow$ gpu_decoded($leafs$, D_t)
    gpu_ray_casting_and_compositing(dec_bricks_tex3D)
  end if
end for

4.1. Adaptive loading and rendering scheme

The process followed to render a single time-step is shown in Algorithm 1.

As each frame is encoded independently into different octrees of bricks, rendering uses only per-frame information. We first start subdividing the octree until the point in which the number of nodes contained in the subtree is such that decoded data can fit in GPU memory, thus computing a forest of octrees. We then traverse front-to back each octree root of the current time-step 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 [9, 11] 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

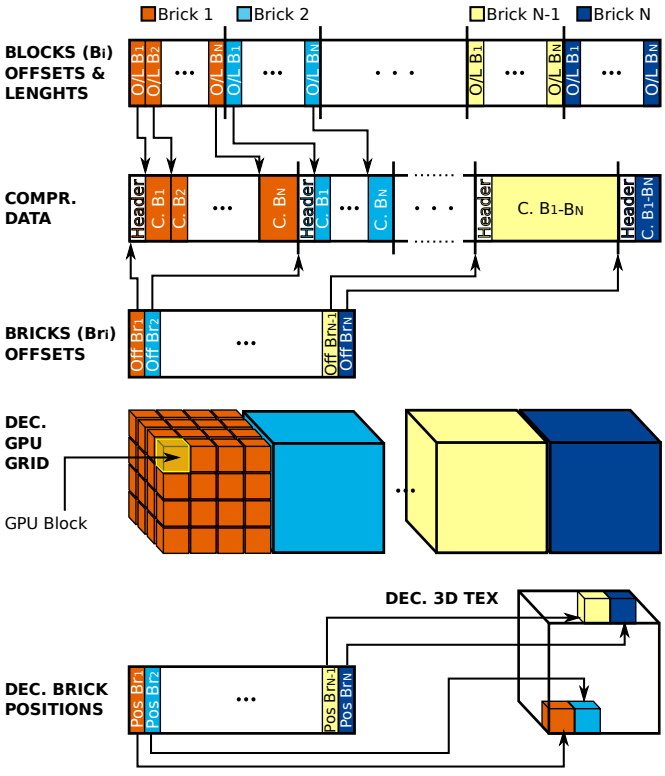

Fig. 4. *Memory layout for GPU decoding.*

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). Decompression uses a thread per decoded voxel, with a grid size equal to the block size. The kernel work is subdivided in two stages: fetching data to shared memory and 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 header data, including 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. Each participating thread always loads 32 bits at a time, and thus decodes to GPU shared memory groups of two index-coefficient pairs in order to avoid bank conflicts. Coefficient dequantization is performed before storing coefficients in shared memory. The index coefficient pairs are ordered so that the first and the second ones are relative to the max and min block coefficients, loaded by the first thread using a single 32-bit load. These coefficients are uniformly dequantized in the range stored in the brick header using the value bits, plus the extra 7 bits in the block header. The remaining coefficients, instead, use only their value bits for dequantization, but in the range of the coefficients of the block, determined when decoding the first two coefficients. After thread synchronization, decompressing a voxel is then just a matter of loading its corresponding values from the indexed dictionary words and linearly combining them with the coefficients present in shared memory.

5. Implementation and results

Our approach has been implemented as an experimental software running on Arch Linux using C++, OpenGL and NVIDIA CUDA 10. It has been tested with a variety of time-varying massive volumetric datasets.

In this paper, we discuss the results obtained with three representative massive time-varying datasets from the JHU Turbulence database [51]: the velocity magnitude field of a forced isotropic turbulence simulation (*ISO*, 1024 time steps 1024^3, float, 4TB), the density field of a homogeneous buoyancy-driven turbulence simulation (*HBDT*, 1010 time steps 1024^3, float, 4TB), and the x-component of the velocity field of a channel flow simulation (*CHAN*, 4000 time steps $2048 \times 512 \times 1536$, float, 24TB). All the benchmark datasets are publicly available and the selected fields were used in our previous work, which presented detailed comparisons with current state-of-the-art solutions in the area of real-time volume rendering from compressed data [5], thus providing a solid context for the results discussed here. Note that the results presented in Diaz et al. [6] use, instead, the pressure field for all the datasets.

5.1. Compression setup

We evaluated our compression strategy by running a battery of tests on the selected time-varying datasets. In all our tests, we used a K-SVD block size $M = 6$, a brick size of $B = 36$ voxels, a dictionary size $K = 2048$ stored in floating point format, and varied the target bit-rate to 0.38 bps, 0.45 bps, and 0.60 bps. Bit-rates were selected so as to be small enough to support real-time rendering, as well as achievable with other fixed-rate codecs.

In order to provide a context for the evaluation of our work, we compare our results with a fixed-rate version of the same codec, as well as with a recently introduced solution based on Elastic Sparse Coding (ESC) [5], 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. [5] have already compared ESC with the major real-time and non-real-time codecs, including COVRA [11], ASTC [52], Hierarchical Vector Quantization (HVQ) [22], CudaCompress wavelet codec (CC) [13], ZFP [27], and SZ [53], we do not repeat all the benchmarks here. As a reference, we include here only the results obtained with the ZFP [27] codec (V 0.5.4), which is a de-facto standard for compression of floating-point volumetric data, and with ASTC (version 10/2017 [54]), which was the best-performing competing real-time codec in previously presented benchmarks [5]. 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. For ASTC, we selected the maximum quality and lowest bit-rate achievable.

5.2. Hardware configuration

Performance has been measured on a single Arch Linux PC with 128GB RAM, a 20-core i9-7900X 3.30 GHz CPU and a GeForce RTX 2080Ti GPU with 11GB VRAM. The uncompressed data is stored on a Synology RS3617Xs+ with a RAID 5 composed of SEAGATE ST10kNE004-1ZF101 disks (BTRFS file system) connected to the processing and rendering machine using a 10 Gbit/s link. The compressed data is stored instead on a local Samsung 9160 Pro 1TB SSD for storage formatted with ext4.

5.3. Compression speed

All the three benchmark datasets were compressed using the incremental approach described in Sec. 3, using a batch size of $BS = 32$ frames, $I = 100$ training iterations for the keyframe, $I = 5$ iterations for the other frames, a coreset for training of $C_{train} = 16Mvox$ and a coreset for tolerance estimation of $C_{eval} = 8MVox$. The training process first downloads the frame from the remote storage to the local SSD, then performs all the other operations on SSD (filtering for construction of the octree hierarchy, coreset extractions, dictionary learning, and final encoding). The bottom-up filtering phase, also including the extraction of the two coresets, is dominated by the initial data transfer time from the file server to the processing node, and takes 20 s/frame for *ISO* and *HBDT* and 32 s/frame for *CHAN*.

Dictionary learning time is independent from dataset size and only very slightly dependent on sparsity due to the combination of coresets with our incremental training approach (10s/frame).

Determining tolerances is also independent from the dataset size, and takes just 4s/frame. Given the computed dictionary, encoding time is linear with the dataset size, and grows sub-linearly with the target sparsity. *ISO* and *HBDT* require 156-181s for the encoding, while *CHAN* requires 238-271s, lower times being for higher compression rates. Thanks to our optimizations, thus, dictionary computation becomes negligible with respect to the other phases.

The encoding speed of the other benchmarked solutions widely varies. ESC, also based on sparse coding, takes 147-185s for dictionary learning, due to not using incremental dictionary refinement, and also has increased encoding time due to per-page size optimization (235-273s for *ISO* and *HBDT*, and 370-415s for *CHAN*). Instead, ZFP is much faster (45-49 s/frame for the overall processing of *ISO* and *HBDT*, 55-76s for *CHAN*), while ASTC is much slower (over 12 min/frame for *ISO*, 15 min/frame for *HBDT*, 17 min/frame for *CHAN*).

	MTV-VAR		**MTV-FIX**		**ESC**		**ASTC**		**ZFP**	
	bps	**PSNR**	**bps**	**PSNR**	**bps**	**PSNR**	**bps**	**PSNR**	**bps**	**PSNR**
ISO	0.38	44.62	0.38	44.06	0.38	46.09			0.34	31.80
	0.45	46.19	0.45	45.65	0.45	47.34			0.45	36.54
	0.60	48.56	0.60	47.99	0.60	48.72	0.59	45.85	0.59	39.84
HBDT	0.38	39.94	0.38	37.03	0.38	39.07			0.34	13.56
	0.45	41.48	0.45	38.71	0.45	39.71			0.45	20.95
	0.60	43.44	0.60	40.97	0.60	40.01	0.59	38.02	0.59	27.29
CHAN	0.38	48.10	0.38	46.31	0.38	48.59			0.34	32.01
	0.45	49.78	0.45	47.93	0.45	49.75			0.45	36.83
	0.60	52.25	0.60	50.29	0.60	50.99	0.59	48.20	0.59	40.21

Table 2. ***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) [5], ASTC [52], and ZFP [27] 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. For ASTC, we selected the maximum quality and lowest bit-rate achievable.**

5.4. Compression rate and distortion

Table 2 summarizes the compression performance of the evaluated codecs at similar bit rates. The results are reported for a single selected frame (number 256 for all datasets). In order to compare with other single-resolution methods, we report the results obtained only for the leaf-level full-resolution grid. Compression rate 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 (MTV) with three different bit-rates using the variable-rate encoder, as well as with a fixed-rate version of the encoder, where the target sparsity has been set to match the same bit rate. Parameters for ESC, ASTC, and ZFP have been also set to provide a similar compression. Empty cells in the table correspond to non-achievable results.

The new variable-rate codec scales well and provides considerable improvement in terms of PSNR with respect to the fixed-rate solution for most of the datasets. The codec is also competitive or on par with respect to ESC, which optimizes rate-distortion within fixed-size pages. 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. ASTC also provides considerably lower quality at comparable bit-rates, and cannot compress data below 0.59bps. This is compatible with the findings of Marton et al. [5].

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 provides an illustration of the visual quality obtained for an isosurface rendering of our benchmark datasets at high magnification, and numerically assesses the visual quality provided by the various encoding schemes by using the Structural Similarity metric [55], which is known to have a good correlation with perceptual quality. The results, shown for a 0.45bps for all schemes except ASTC and compressed at 0.59bps, show that the sparse-coding schemes (MTV and ESC) provide better visual quality than ZFP and ASTC, even if ASTC requires over 30% more storage. Variable-rate schemes also provide significantly better quality than fixed-rate ones, with SSIM values that increase by 7-18% depending on the dataset, which are considered noticeable differences.

The behavior in terms of error distribution of the various real-time codecs is illustrated in Fig. 6. The images show, using color coding, the average error computed along the z direction for the three data sets, collapsing errors on the 3D volumes to a 2D image. It can be notices that in all cases error increases at block boundaries, since all the techniques use block-based compression in order to support parallel real-time decoding with random access. Deblocking techniques, orthogonal to the work covered here, could be applied to further improve visual quality, at the expense of decoding and/or rendering time [56]. The error images also show that the tolerance-driven codec proposed in this paper successfully maintains a near-constant quality, while the other codecs have a much more variable distribution, with low-error blocks mixed with high-error ones. Moreover, the ESC codecs clearly shows page boundary artifacts, especially visible in the CHAN dataset, which has significant differences between the boundary region, with high turbulence, and the central one with laminar flow.

5.5. Interactive exploration

We evaluated the rendering performance of our framework on a number of interactive inspection sequences of the three dynamic datasets tested, using the 0.45bps configuration shown in Table. 2. This leads to the creation of compressed octree representations of 93.6 GB (*ISO*), 92.7 GB (*HBDT*), and 551.7 GB (*CHAN*). The individual frame size is of 93.6MB for *ISO* and *HBDT* and of 141.2MB for *CHAN*.

The qualitative performance of our adaptive GPU ray-caster, loading data from local SSD disk and rendering with a RTX 2080Ti GPU is illustrated in the accompanying video. All images have been recorded on a window size of 1920×1080 pixels using a 1 voxel/pixel accuracy. Representative frames are depicted in Fig. 1 and Fig. 7.

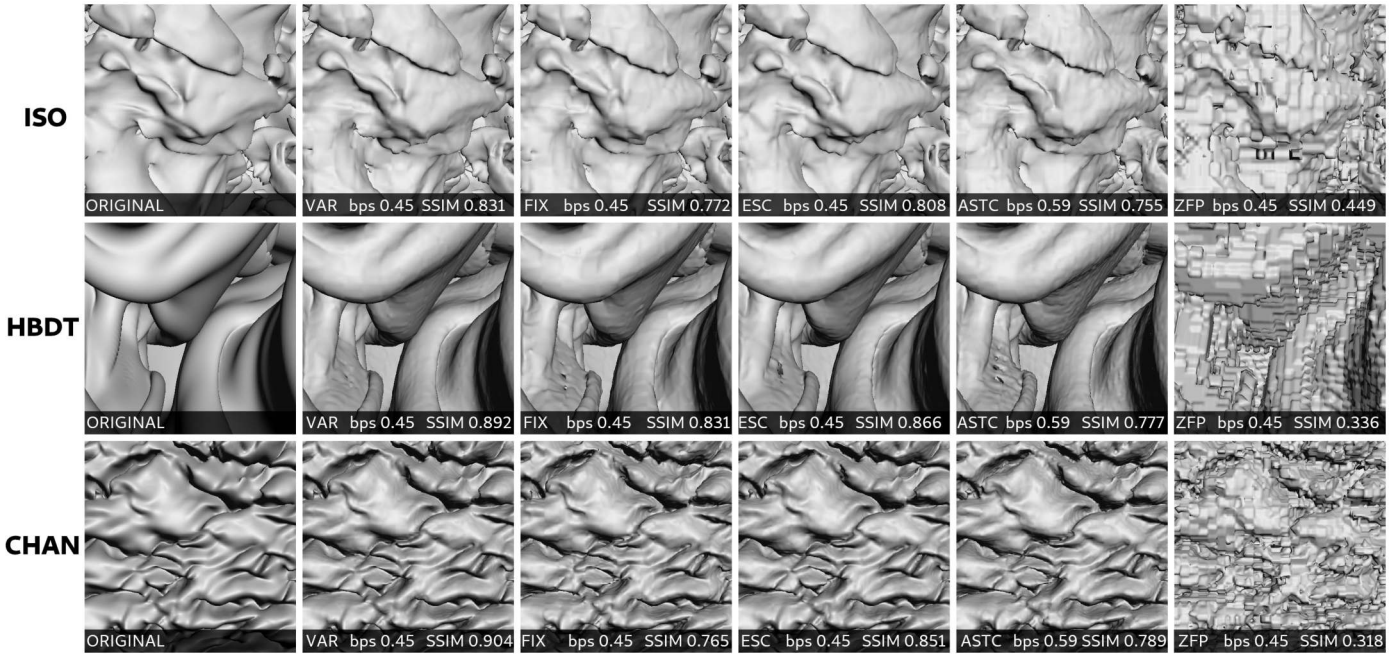

Fig. 5. ***Perceptual quality assessment.*** **Isosurface rendering and corresponding SSIM (structural similarity) values of the rendered images: from top to bottom ISO, HBDT and CHAN. All encodings use 0.45 bps except ASTC which is encoded at 0.59 bps (minimum reachable bps with this coder).**

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 the targeted 10Hz, varying from 15Hz for medium-range closeups where most of the full dataset is visible at high resolution to well above 40Hz for overall views or extreme closeup views of models, where we can better exploit level-of-detail and view culling to reduce uploading, decoding and rendering overhead. In particular, for the sequences included in the accompanying video, the average frame rate was of 33.0Hz for ISO (min 5.9Hz, max 120.6Hz), 37.2Hz for HBDT (min 6.4Hz, max 123.7Hz), and 24.9Hz for CHAN (min 7.9Hz, max 48.9Hz). The absolute worst case occurs when all the model is visible with a semi-transparent transfer function that forces, at the same time, full decoding and full traversal.

Data loading is performed in parallel with the decoding and rendering kernels. GPU profiling reveals that, on average, half of the frame time is devoted to rendering, while the other half is occupied by all other operations: cut identification, culling, data loading, and decompression. In particular, the raycasting kernel takes on average from 35% (CHAN) to 58% (ISO and HBDT) of the total frame time. This is not surprising, since the goal of this paper is to show the feasibility of using a custom compressed representation for cache-less full-frame decoding during real-time spatio-temporal exploration, and no particular optimization has been made to reduce the number of decoded blocks, beyond plain view-frustum and transfer-function culling. The introduction of caching when exploring static frames and occlusion culling for improving performance on opaque models would result in increased average performance, and could be done exactly as in Marton et al. [5]. In this work, we focus instead on just measuring typical worst-case situations.

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 due to partial refinements are visible in the animation. Such a streaming architecture is, however, limited 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 real-time-decodable 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

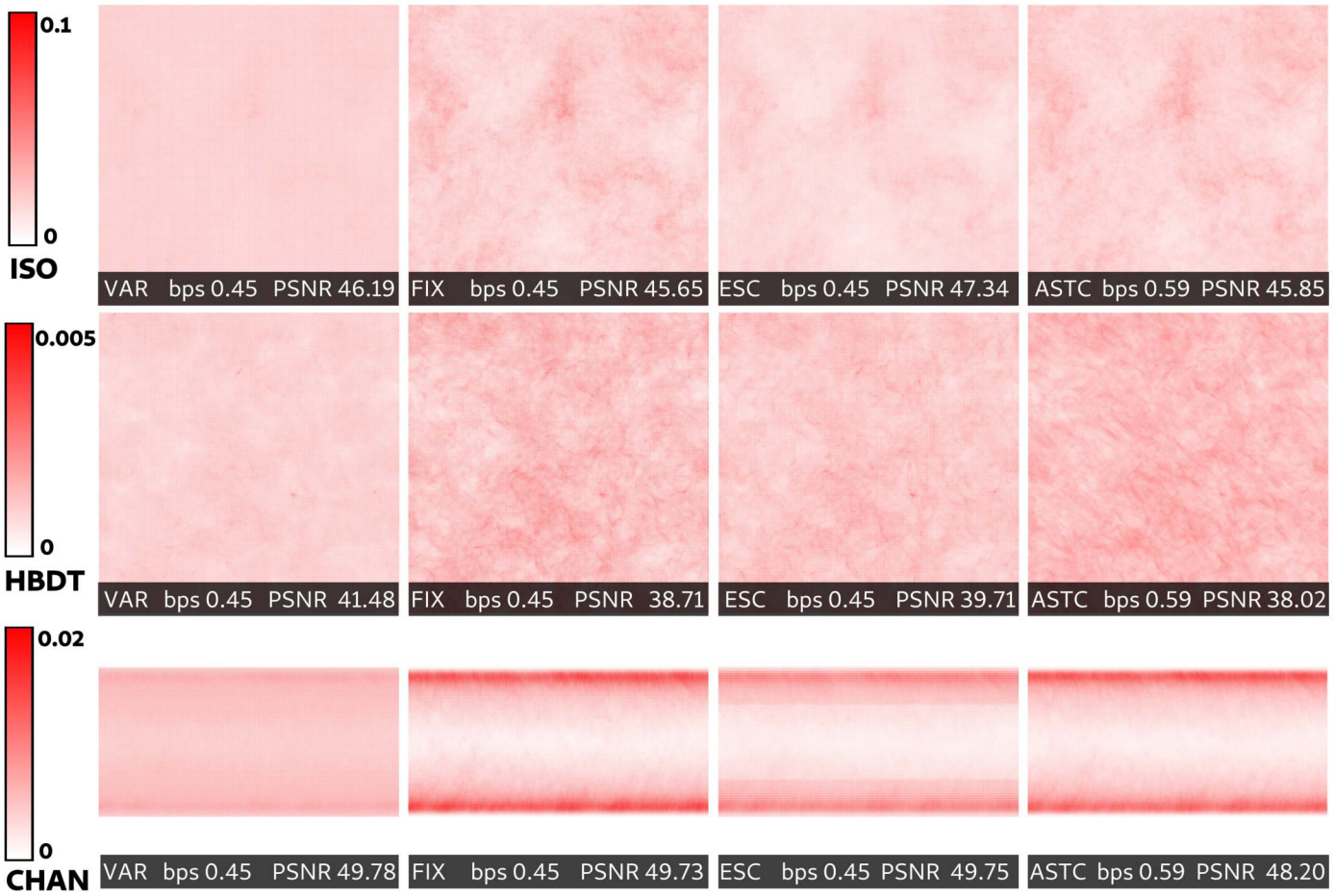

Fig. 6. ***Error distribution maps.*** **Accumulated error maps on the Z direction for all datasets: from top to bottom ISO, HBDT and CHAN. All encodings use 0.45 bps except ASTC which is encoded at 0.59 bps (minimum reachable bps with this coder). ISO and HBDT show the full 1024x1024 projection, while CHAN shows the central crop of 1024x512.**

CFD simulation. Fig. 8 and the accompanying video illustrate our preliminary results, which show the possibility of exploring in real-time the vorticity field around buildings at urban scales.

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 projects VIGECLAB and TDM (POR FESR 2014-2020 Action 1.2.2).

References

[1] Weiss, K, Floriani, L. Modeling and visualization approaches for time-varying volumetric data. In: Proc. Advances in Visual Computing. 2008, p. 1000–1010.

[2] She, B, Boulanger, P, Noga, M. Real-time rendering of temporal volumetric data on a GPU. In: Proc. IEEE InfoVis. 2011, p. 622–631.

[3] Li, Y, Perlman, E, Wan, M, Yang, Y, Meneveau, C, Burns, R, et al. A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence. Journal of Turbulence 2008;9.

[4] Irion, R. The terascale supernova initiative: Modeling the first instance of a star's death. SciDAC Review 2006;2(1):26–37.

[5] Marton, F, Agus, M, Gobbetti, E. A framework for gpu-accelerated exploration of massive time-varying rectilinear scalar volumes. Computer Graphics Forum 2019;38(3).

[6] Díaz, J, Marton, F, Gobbetti, E. MTV-Player: Interactive spatio-temporal exploration of compressed large-scale time-varying rectilinar scalar volumes. In: Proc. STAG. 2019, p. 1–10.

[7] Balsa Rodriguez, M, Gobbetti, E, Iglesias Guitián, J, Makhinya, M, Marton, F, Pajarola, R, et al. State-of-the-art in compressed GPU-based direct volume rendering. Computer Graphics Forum 2014;33(6):77–100.

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

[9] Crassin, C, Neyret, F, Lefebvre, S, Eisemann, E. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: Proc. I3D. 2009, p. 15–22.

[10] Engel, K. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In: Proc. IEEE LDAV. 2011, p. 123–124.

[11] 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 2012;31(3/4):1315–1324.

[12] Reichl, F, Treib, M, Westermann, R. Visualization of big SPH simulations via compressed octree grids. In: Proc. IEEE Big Data. 2013, p. 71–78.

[13] Treib, M, Burger, K, Reichl, F, Meneveau, C, Szalay, A, Westermann, R. Turbulence visualization at the terascale on desktop PCs. IEEE TVCG 2012;18(12):2169–2177.

[14] Hadwiger, M, Beyer, J, Jeong, WK, Pfister, H. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. IEEE TVCG 2012;18(12):2285–2294.

[15] Fogal, T, Schiewe, A, Kruger, J. An analysis of scalable GPU-based ray-guided volume rendering. In: Proc. IEEE LDAV. 2013, p. 43–51.

[16] Delp, E, Mitchell, O. Image compression using block truncation coding.

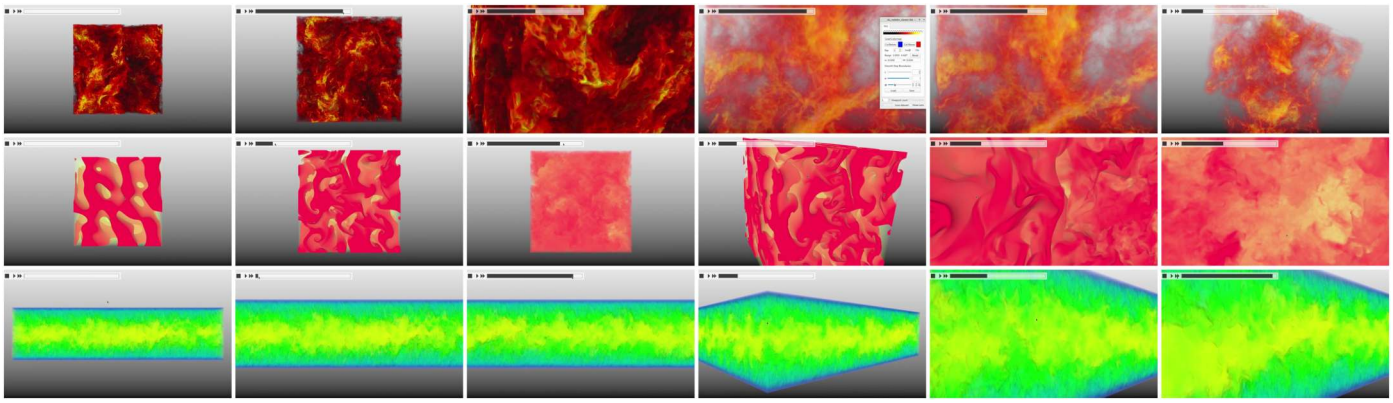

Fig. 7. ***Representative frames of the accompanying video.*** **Our rendering architecture supports interactive spatial exploration, modification of the transfer function parameters, playing the sequence forwards and backwards at various speeds, and full random access to individual frames.**

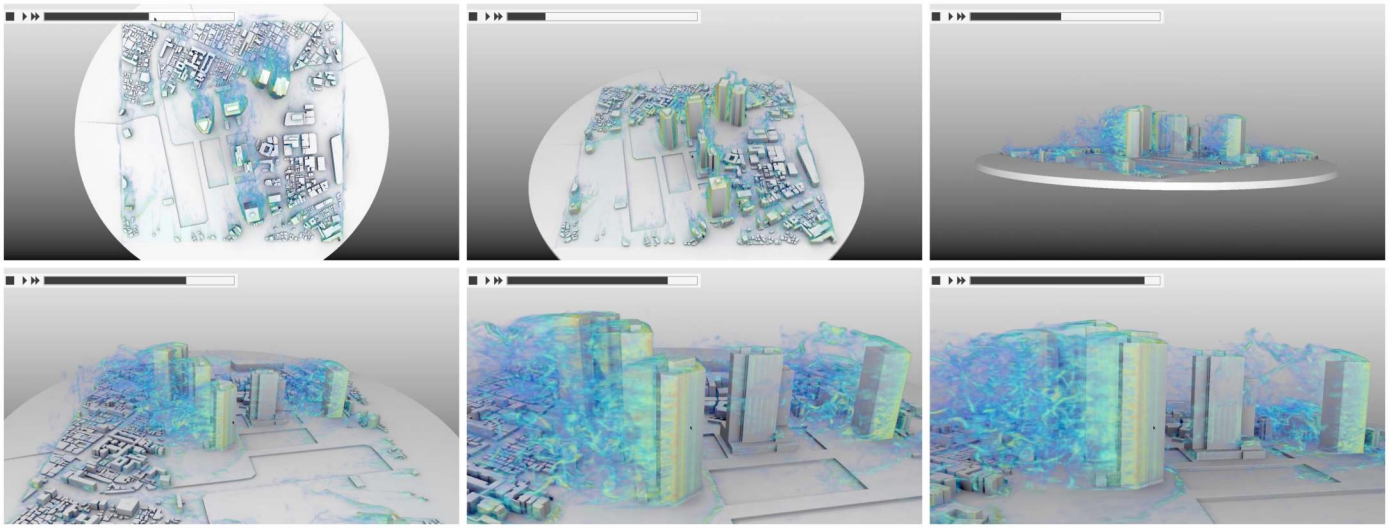

Fig. 8. ***Example of application to urban CFD simulation.*** **Frames from an interactive sequence of spatio-temporal exploration of the vorticity field around buildings at a urban scale (Shinjuku, Japan).**

IEEE Trans Comm 1979;27(9):1335–1342.
[17] Craighead, M. Gl_nv_texture_compression_vtc. OpenGL Extension Registry; 2004.
[18] Yela, H, Navazo, I, Vazquez, P. S3Dc: A 3Dc-based volume compression algorithm. Computer Graphics Forum 2008;:95–104.
[19] Iglesias Guitián, JA, Gobbetti, E, Marton, F. View-dependent exploration of massive volumetric models on large scale light field displays. The Visual Computer 2010;26(6–8):1037–1047.
[20] Fout, N, Ma, KL. Transform coding for hardware-accelerated volume rendering. IEEE TVCG 2007;13(6):1600–1607.
[21] Parys, R, Knittel, G. Giga-voxel rendering from compressed data on a display wall. In: Proc. WSCG. 2009, p. 73–80.
[22] Schneider, J, Westermann, R. Compression domain volume rendering. In: Proc. IEEE Vis. 2003, p. 293–300.
[23] Kraus, M, Ertl, T. Adaptive texture maps. In: Proc. Graphics Hardware. 2002, p. 7–15.
[24] Guthe, S, Goesele, M. Variable length coding for GPU-based direct volume rendering. In: Proc. VMV. 2016, p. 77–84.
[25] 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 2017;35:8–18.
[26] Aharon, M, Elad, M, Bruckstein, A. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. IEEE TSP 2006;54(11):4311–4322.
[27] Lindstrom, . Fixed-rate compressed floating point arrays. IEEE TVCG 2014;20(12):2674–2683.
[28] 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. In: Multi-Modality Imaging. 2018, p. 191–215.
[29] Suter, S, Iglesias Guitián, J, Marton, F, Agus, M, Elsener, A, Zollikofer, C, et al. Interactive multiscale tensor reconstruction for multiresolution volume visualization. IEEE TVCG 2011;17(12):2135–2143.
[30] Ballester-Ripoll, R, Lindstrom, P, Pajarola, R. TTHRESH: Tensor compression for multidimensional visual data. arXiv preprint arXiv:180605952 2018;.
[31] Park, J, Gutenko, I, E. Kaufman, A. Transfer function-guided saliency-aware compression for transmitting volumetric data. IEEE Transactions on Multimedia 2017;PP:1–1.
[32] Shen, HW, Johnson, CR. Differential volume rendering: A fast volume visualization technique for flow animation. In: Proc. IEEE Vis. 1994, p. 180–187.
[33] Guthe, S, Straßer, W. Real-time decompression and visualization of animated volume data. In: Proc. IEEE Vis. IEEE; 2001, p. 349–572.
[34] Lum, EB, Ma, KL, Clyne, J. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. IEEE TVCG 2002;8(3):286–301.
[35] Woodring, J, Wang, C, Shen, HW. High dimensional direct rendering of time-varying volumetric data. In: Proc. IEEE Vis. 2003, p. 417–424.
[36] Wang, H, Wu, Q, Shi, L, Yu, Y, Ahuja, N. Out-of-core tensor approximation of multi-dimensional matrices of visual data. ACM TOG 2005;24(3):527–535.

[37] Westermann, R. Compression domain rendering of time-resolved volume data. In: Proc.IEEE Vis. 1995, p. 168–175.
[38] Ma, KL, Shen, HW. Compression and accelerated rendering of time-varying volume data. In: Proc. International Workshop on Computer Graphics and Virtual Reality. 2000, p. 82–89.
[39] Wang, C, Gao, J, Li, L, Shen, HW. A multiresolution volume rendering framework for large-scale time-varying data visualization. In: Proc. Volume Graphics. 2005, p. 11–19.
[40] Shen, HW. Visualization of large scale time-varying scientific data. Journal of Physics 2006;46(1):535–544.
[41] Ko, CL, Liao, HS, Wang, TP, Fu, KW, Lin, CY, Chuang, JH. Multiresolution volume rendering of large time-varying data using video-based compression. In: Proc. IEEE Pacific Vis. 2008, p. 135–142.
[42] Mensmann, J, Ropinski, T, Hinrichs, K. A GPU-supported lossless compression scheme for rendering time-varying volume data. In: Proc. Volume Graphics. 2010, p. 109–116.
[43] Wang, C, Yu, H, Ma, KL. Importance-driven time-varying data visualization. IEEE TVCG 2008;14(6):1547–1554.
[44] Jang, Y, Ebert, DS, Gaither, KP. Time-varying data visualization using functional representations. IEEE TVCG 2012;18(3):421–433.
[45] Nagayasu, D, Ino, F, Hagihara, K. Two-stage compression for fast volume rendering of time-varying scalar data. In: Proc. GRAPHITE. 2008, p. 275–284.
[46] Wang, C, Yu, H, Ma, KL. Application-driven compression for visualizing large-scale time-varying data. IEEE CGA 2010;30(1):59–69.
[47] Cao, Y, Wu, G, Wang, H. A smart compression scheme for GPU-accelerated volume rendering of time-varying data. In: Proc. IEEE ICVRV. 2011, p. 205–210.
[48] Pulido, J, Livescu, D, Kanov, K, Burns, RC, Canada, C, Ahrens, JP, et al. Remote visual analysis of large turbulence databases at multiple scales. J Parallel Distrib Comput 2018;120:115–126.
[49] Rubinstein, R, Zibulevsky, M, Elad, M. Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit. Tech. Rep.; CS Technion; 2008.
[50] Efraimidis, PS. Weighted random sampling over data streams. In: Algorithms, Probability, Networks, and Games. 2015, p. 183–195.
[51] JHU, . Johns Hopkins Turbulence Databases. `http://turbulence.pha.jhu.edu/datasets.aspx`; 2016. [accessed: 2018-10-31].
[52] Nystad, J, Lassen, A, Pomianowski, A, Ellis, S, Olson, T. Adaptive scalable texture compression. In: Proc. HPG. 2012, p. 105–114.
[53] Di, S, Cappello, F. Fast error-bounded lossy HPC data compression with SZ. In: Proc. IEEE IPDPS. 2016, p. 730–739.
[54] ASTC compression library. `https://github.com/ARM-software/astc-encoder`; 2017. [accessed: 2018:10:31].
[55] Wang, Z, Bovik, A, Sheikh, H, Simoncelli, E. Image quality assessment: from error visibility to structural similarity. IEEE TIP 2004;13(4):600–612.
[56] Marton, F, Iglesias Guitián, J, Diaz, J, Gobbetti, E. Real-time deblocked GPU rendering of compressed volumes. In: Proc. VMV. 2014, p. 167–174.