Eurographics Conference on Visualization (EuroVis) 2019
M. Gleicher, H. Leitte, and I. Viola
(Guest Editors)

Volume 38 (2019), Number 3

A framework for GPU-accelerated exploration of massive time-varying rectilinear scalar volumes

Fabio Marton[1], Marco Agus[1,2], and Enrico Gobbetti[1]

[1]CRS4, Italy
[2]KAUST, Saudi Arabia

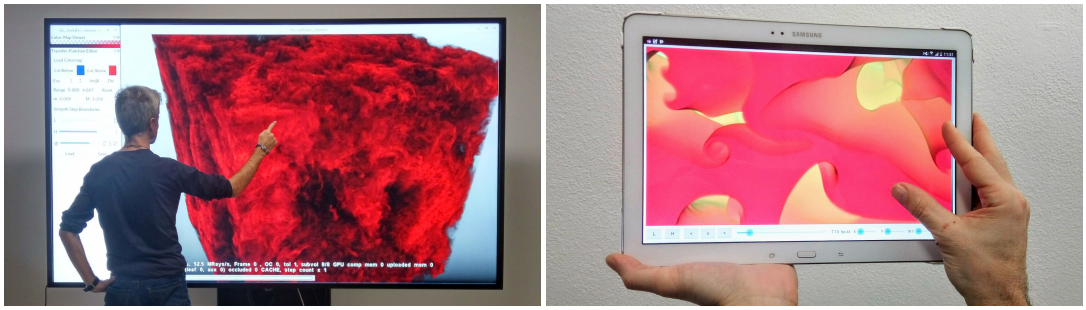

Figure 1: ***Fat and thin client.*** *Our flexible framework supports fully interactive spatiotemporal exploration of simulations with thousands of multi-billion-voxel frames. Left: fat client performing local rendering on a 1920x1080 touch screen driven by a single graphics PC with NVIDIA GTX 1080Ti. Right: thin client on Samsung Galaxy Note Pro SM-P905 Android tablet connected to a remote rendering server.*

Abstract

We introduce a novel flexible approach to spatiotemporal exploration of rectilinear scalar volumes. Our out-of-core representation, based on per-frame levels of hierarchically tiled non-redundant 3D grids, efficiently supports spatiotemporal random access and streaming to the GPU in compressed formats. A novel low-bitrate codec able to store into fixed-size pages a variable-rate approximation based on sparse coding with learned dictionaries is exploited to meet stringent bandwidth constraint during time-critical operations, while a near-lossless representation is employed to support high-quality static frame rendering. A flexible high-speed GPU decoder and raycasting framework mixes and matches GPU kernels performing parallel object-space and image-space operations for seamless support, on fat and thin clients, of different exploration use cases, including animation and temporal browsing, dynamic exploration of single frames, and high-quality snapshots generated from near-lossless data. The quality and performance of our approach are demonstrated on large data sets with thousands of multi-billion-voxel frames.

CCS Concepts

• ***Human-centered computing*** → ***Scientific visualization;*** • ***Computing methodologies*** → ***Computer graphics;*** *Graphics systems and interfaces;*

1 Introduction

Medical, scientific and engineering applications routinely generate time-varying rectilinear scalar volumes with thousands of time steps and billions of voxels per frame [LPW*08, Iri06, Tur]. While a number of efforts have been proposed to generate reduced abstract representations of entire animations [WYM08, JEG12, FE17], interactive visual exploration of whole datasets is also crucial to understand many phenomena in scientific and technological fields [WF08, SBN11]. The need for interactivity in space, time, and rendering parameters imposes current visualization applications, typically running on desktop PCs or other personal devices, to meet stringent memory and timing constraints. Meeting such constraints within a real-time application is extremely difficult given the sheer size of the explored data and the complexity of volumetric rendering. This has led researchers to introduce a variety of specialized adaptive approximated visualization techniques and frameworks that trade quality with memory and speed (see Sec. 2). Such methods must support, in practice, a wide variety of data exploration scenarios with very different temporal and spatial fidelity constraints. These include exploring animations while changing camera and

rendering settings in order to perceive dynamic effects, quick non-linear temporal browsing to identify interesting time-steps, as well as analyzing a selected time-step to perceive the finest shape details. Moreover, while simulations are computed on large parallel machines and stored on servers, their results are visually explored on graphics PCs, workstations, or mobile setups [NJ16]. We therefore need to move massive amounts of data efficiently from remote storage to local storage and graphics hardware, and also to handle rendering scalability issues.

In recent years, many GPU-based real-time direct volume rendering (DVR) architectures have been developed. They employ multiresolution data representations, compression, out-of-core methods and data streaming to enable interactive visualization of massive volumetric datasets. While these architectures have been extremely successful in the exploration of static datasets [TBR*12, BRGIG*14, BHP15], current techniques do not fully support real-time exploration of dynamic data with full spatial and temporal control (see Sec. 2). In particular, in order to cope with bandwidth limitations and limited decoding speed, many of the present real-time techniques are forced to amortize the updates of a rendering working-set over multiple frames. This approach, however, reduces the required bandwidth, but introduces the unwanted result of mixing real dynamic effects with spurious effects created by incremental updates.

In this paper, we introduce a novel flexible approach that exploits multiple compressed representations within a GPU-accelerated compression-domain configurable renderer to support a variety of exploration means for large static and dynamic rectilinear scalar field volumes (Sec. 3). In our approach, an input time-varying rectilinear scalar field is converted off-line into a GPU-friendly out-of-core data structure consisting of a time sequence of per-frame multiresolution pyramids with a dual near-lossless and lossy representation. The near-lossless representation is employed to support high-quality static frame rendering, while the lossy representation is specially tuned for high-speed on the-fly decoding. Extending previous fixed-rate sparse approximation approaches [GIM12], a novel bit allocation algorithm approximates a multiple-choice knapsack problem to fit in fixed-size pages sparse variable-length linear combinations of prototype blocks stored in a learned overcomplete dictionary. Each resolution level is organized in a hierarchically tiled 3D grid organizing the data into pages of non-overlapping bricks of voxel blocks, supporting spatiotemporal random access and streaming to the GPU in compressed formats (Sec. 4). A flexible rendering framework mixes and matches GPU kernels for seamless support of different exploration use cases, exploiting our data layout to efficiently perform parallel object-space and image-space operations (Sec. 5). Animation and temporal browsing without temporal artifacts is achieved by a cache-less adaptive GPU raycasting algorithm, which streams to GPU a conservatively identified working set of the lossy representation, using transient and fast decoding for compression-domain rendering. Dynamic exploration of single frames is achieved by incorporating a ray-guided streaming process supporting visibility feedback and incremental refinement from the lossy representation, as well as high-quality snapshots generated from the near-lossless ones when the camera stops moving. The framework supports the creation of fat clients running the full rendering algorithm, as well as thin clients receiving image data streamed from a rendering server (Sec. 6). As a result, our approach can support different exploration use cases on fat and thin clients. These include animation and temporal browsing without temporal artifacts of datasets with thousands of multi-billion-voxel frames, dynamic exploration of single frames, and high-quality snapshots from near-lossless data (Sec. 7).

2 Related Work

Visualizing massive datasets requires scalable solutions in terms of data representation, data/work partitioning and work/data reduction. We briefly discuss here the methods that are most closely related to ours. For a wider coverage, we refer the reader to established surveys on modeling and visualization approaches for time-varying volumetric data [WF08], compression-domain DVR [BRGIG*14], GPU-based large-scale DVR [BHP15], and mobile DVR [NJ16].

Output-sensitive approaches require adaptive loading of compressed data stored in out-of-core structures, which range, for static datasets, from a single-resolution set of compressed bricks [TBR*12] to multiresolution structures such as octrees [CNLE09, Eng11, GIM12, RTW13] or hierarchical grids of bricks [HBJP12, FSK13]. In this context, a scalable preprocessing method, compressed streaming, and fast on-demand spatially independent decompression on the GPU, are required for maximum benefits [FM07]. The simplest hardware-supported fixed-rate schemes [YNV08, IGM10, NLP*12] support general random access but present a limited flexibility in terms of achievable compression and supported data formats. Different methods based on vector quantization along with adaptive texture maps [KE02, GG16, YZW*17], Laplacian pyramid compression techniques [SW03], or wavelet-based transform coders [FM07, PK09] have also been proposed, but quality and compression ratios are limited by the dictionary size of the vector quantization stage, which forces the result to contain a limited amount of different blocks [Ela08], making these techniques more applicable to low dynamic range data (e.g., 8-16 bit integers). An alternative to vector quantization is tensor approximation [SIM*11, BRLP18], which is based on a rank reduction of a data-specific preferential basis. Decompression, however, cannot be performed at interactive rates for full time steps. Another alternative is the recent ZFP floating point compressor [Lin14], which performs better for high-quality compression, and is thus used in our context for high-quality rendering of static frames.

3D volumetric compression is often extended to 4D in order to exploit correlation between time steps [SJ94, GS01, LMC02, ILRS03, WWS03, WWS*05]. The main limitation of these full-4D approaches is the added complexity in moving through time in a non-conventional way (random access, backwards, fast-forward, ...), and the need of having in memory a small set of frames to render a single time-step. Other techniques exploit temporal coherence by encoding each voxel with respect to reference key-frames [Wes95, MS00, WGLS05, She06, KLW*08, MRH10, WYM08, SBN11, JEG12]. Random access to different time steps is simpler, but these methods suffer from similar limitations than the previous ones: since key-frames are needed to decode single time-steps, memory and bandwidth limitations impose restrictions on the size of the data they can deal with. A combination of the previous techniques can be found in many systems (e.g., [FM07, NIH08, WYM10, CWW11]).

Encoding each time step individually by using 3D compression

methods (e.g., [GIM12, TBR*12, PLK*18]) guarantees full random access and is less limiting than in video compression, since dynamic 3D volumes already provide one more dimension with respect to 2D images to exploit for data reduction. In order to achieve good compression, however, very advanced encoders are required at the very low bitrates required by dynamic presentation of volumes. Treib et al. [TBR*12] use a per-time-step wavelet-based bricked representations combined with run-length and entropy encoding. Pulido et al. [PLK*18] employ a similar approach, only streaming the coefficients required for a specific scale to support scale-specific visualization. While excellent signal-to-noise ratios can be obtained, and effective exploration of single time steps is supported, no real-time performance is achieved when advancing in time, since the inverse transformation is relatively costly.

COVRA [GIM12] also employs a sparse representation based on a learned dictionary. While COVRA focuses on exploration of static datasets, our framework significantly differs at the system level. Our paged fixed-size layout improves over COVRA's octree representation by supporting brick access and brick spatial location identification through index computation rather than structure traversal (Sec. 4), making it possible to efficiently write rendering kernels implementing parallel object-space and image-space operations (see Sec. 5), and to mix-and-match them to support a variety of exploration use cases. Similarly to COVRA, we learn the dictionary by applying the K-SVD algorithm [AEB06] to a small weighted randomized subset of volume blocks. The advantage of such sparse-coding approach is that reconstruction is achieved by a simple linear combination of blocks, which can be computed very fast on current GPUs. In terms of compressed representation, we improve over COVRA's fixed-rate scheme through a novel constrained variable-rate encoder and a better bit allocation scheme, increasing rate-distortion performance, while still producing fixed-size pages for optimized I/O and memory management. Moreover, our scheme produces non-overlapping bricks, with significant compression gain with respect to the bricks with apron of COVRA and related approaches, which maintain as an extra voxel border around each brick, and duplicate the values across brick boundaries. Finally, COVRA relies on non-conservative incremental updates and visualizes small dynamic datasets by fully decoding and pre-caching them, while our framework is designed to support complete dynamic updates from unlimited-length time sequences stored out-of-core.

3 Overview

Our flexible architecture strives to permit the implementation of interactive exploration systems that support fully dynamic spatiotemporal changes, as well as accurate still-frame inspection. It is made of general processing and rendering components which can be mixed and matched to implement the required inspection features. The overall structure of a typical configuration is depicted in Fig. 2.

The input data for our process is a series of T rectilinear scalar volumes of dimension $N_x \times N_y \times N_z$. Our out-of-core representation, based on per-frame levels of hierarchically tiled 3D grids, supports spatiotemporal random access and streaming to the GPU in compressed formats. Each frame, encoded independently of all others, has a dual representation based on two parallel multiresolution structures of compressed data: a highly compressed lossy structure and a near lossless one (Sec. 4). At run-time the highly compressed structure is used for rendering during interaction or animation, so that the limited available bandwidth can be used for temporally variable data. The near-lossless representation is used, instead, for high quality rendering of still frames. Both renderers are performed by a flexible rendering framework which mixes and matches GPU kernels for seamless support of different exploration use cases (Sec. 5). Moreover, the high-bandwidth representation is typically maintained on a remote server, while the low-bandwidth representation can be optionally transmitted to a fat client for local high-speed rendering. At the same time, the low-bandwidth representation can be used on the server to provide interactive image streams for thin-clients, as those, e.g., implemented on mobile devices (Sec. 6).

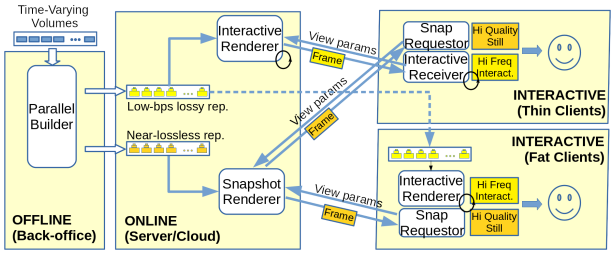

Figure 2: ***Architecture overview.*** *A near-lossless and a low-bitrate representation are built from the input rectilinear scalar volumes. The near-lossless representation is exploited for computing, server side, high quality still frames, while the low-bitrate one is used for high-framerate image generation. Thin clients receive images from a server-side renderer, while fat clients perform local rendering on a locally downloaded copy of low-bitrate data.*

4 Data layout and compressed data representation

In order to support spatial levels of details, each frame is represented by a pyramid of 3D grids, each of which is a progressively lower resolution representation of the same rectilinear volume. Each lower level represents the volume using (approximately, see below) half as many voxels in each dimension.

Fig. 3 illustrates the layout of a single resolution level, which is itself based on a Hierarchically Tiled Array (HTA) [BGH*06] recursive indexing of the 3D voxel grid, with pages composed of P^3 compressed bricks composed of B^3 voxels. Pages represent the main unit for uploading data to the GPU, while bricks are used as unit for all GPU operations. In order to simplify indexing, the extent of each resolution level is adjusted to be a multiple of $P \cdot B$ for each axis. As all pages contain the same number of bricks, and bricks contain the same number of voxels, pages, bricks, and voxels can be addressed using a layout function performing simple index computations. Moreover, it is also possible to invert the indexing, going back from voxel index to brick and page index. This two-level tiling improves locality of reference. All these features are exploited in our rendering kernels (see Sec. 5). Using this layout, forward and backward index computation just requires to know the number of pages per level and the corresponding starting brick offset for each level. This information is precomputed at construction time. Forward and backward conversions can be directly implemented with simple sums, multiplications, div, and mod operations.

For each time-step, we record the range $v_{min}..v_{max}$ of the values contained. Three parallel HTA structures with the same layout are used to represent a frame. The first is a histogram array, which contains for each brick the quantized binary histogram of the values

contained in the brick itself and all its higher resolution versions. The second is the low-bitrate representation (see Sec. 4.2), while the third is the near-lossless representation (see Sec. 4.1)

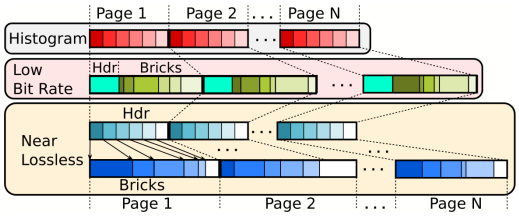

Figure 3: ***General data layout.*** *Three parallel fixed-size HTA structures with the same layout are used to represent a frame. In order to support high-quality encoding the near-lossless representation uses the fixed-size structure as an index to variable-rate representation.*

Processing begins by computing the number of levels required to cover the volume. The compression process, then, proceeds possibly in parallel for each time-step. First, all the levels of the multiresolution pyramid in near-lossless format (see Sec. 4.1) are computed bottom-up and stored on disk in the HTA layout. Then, we proceed to compute the low-bitrate representation, which is done in two steps (see Sec. 4.2), first learning the parameters of the representation, and then iterating over all pages to perform the encoding using the learned parameters. To further speed up the process, the individual pages are encoded in parallel. Parallelization among frames is done using different processes that can be eventually executed on different machines. Parallelization within a single frame is, instead, within the same process, as it is easier to share the output representation and synchronize writing to frame files. At the end of construction, we store on disk a small header containing the information for forward and backward index computation (i.e., the number of levels, the number of pages per level, and the starting brick offset per level).

4.1 Near-lossless compression for high-quality frames

The near-lossless representation must provide a high-quality representation of its contents that is transiently decoded during static frame rendering. Since no strict frame-rate constraints, and therefore bandwidth constraints, must be met for static frame rendering, we adopt a general representation in which each brick is encoded at variable bit rate for meeting quality constraints. In order to ensure random brick indexing, the constant-size HTA of the near-lossless representation just contains a 32-bit pointer to the encoded brick data. The brick data is stored in a separate out-of-core array in the same order of the HTA. Several high-quality encoders can be used for near-lossless representation of bricks. For this paper, we employ the ZFP [Lin14] codec (version 0.5.4 [Lin]). We use the fixed accuracy mode (which usually yields the best compression rates) and vary absolute error tolerance (`-a`).

4.2 Low-bitrate compression for dynamic data presentation

The low-bitrate representation is used for bandwidth-critical operations, which occur when transferring the dataset from a remote server to a local fat client, or when transferring data from storage to graphics memory at each frame during dynamic operations. It is thus essential to ensure maximum compression, considering, e.g., that 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. Moreover, supporting a 10 frames/s animation requires a codec able to decompress and render at least 10 Gvox/s. At such a low bitrate, the techniques able to provide the best approximations, while supporting fast GPU decompression, are based on learned representations [BRGIG*14, BRLP18]. In this work, we solve an optimization problem by fitting into constant-size pages a variable-rate sparse approximation of volume blocks that minimizes the overall error. Constant-size pages allow for implicit indexing, easy I/O and memory management, as well as constant-memory and near-constant-time transfer and decoding time of local volumetric regions. At the same time, our variable-rate approximation leads to low error while supporting fast parallel decoding.

Our representation approximates each block b_i of the volume hierarchy by a sparse linear combination of prototype blocks stored in an overcomplete dictionary $\boldsymbol{D} \in \mathbb{R}^{m \times n}$ learned from the input volume sequence. For this, we map each block b_i of size $m = M^3$ to a column vector $\boldsymbol{y}_i \in \mathbb{R}^m$. The dictionary $\boldsymbol{D}$ is structured to have each prototype block mapped to a column vector $\boldsymbol{d}_k \in \mathbb{R}^m$ of unit length. The first column $\boldsymbol{d}_1$ is assumed to be the constant $\boldsymbol{d}_1 = \frac{\mathbf{1}}{\sqrt{m}}$ and does not participate to training, while the others are maintained at zero mean. Given $\boldsymbol{D}$, our compressed representation for a block $\boldsymbol{y}_i$, thus consists in a set of indices k_i and associated non-zero coefficients $\boldsymbol{\gamma}_i$, such that $\boldsymbol{y}_i \approx \sum_{k=1}^{K_i} \gamma_{ik} \boldsymbol{d}_{k_{ik}}$, where K_i is the number of non-zero coefficients in the representation of block i. In order to fit variable-length representations of the blocks into a fixed page size, we first perform dictionary learning, and then, given the learned dictionary we perform encoding so as to meet our size constraints.

4.2.1 Dictionary learning

First, given the target page size, we compute the average target non-zero count K. The optimal dictionary $\boldsymbol{D}$ is then computed by jointly optimizing the columns $2..n$ of the dictionary and the sparse representation according to the objective function

$$\min_{\boldsymbol{D},\boldsymbol{\gamma}_i} \sum_i w_i \|\boldsymbol{y}_i - \boldsymbol{D}\boldsymbol{\gamma}_i\|_2^2 \text{ subject to } \forall i, \|\boldsymbol{\gamma}_i\|_0 \leq K \qquad (1)$$

where $\|\boldsymbol{\gamma}_i\|_0$ counts the non-zero entries of $\boldsymbol{\gamma}_i$ and w_i is a weight associated to each training sample. For training, we employ a weighted variation of K-SVD [AEB06], which performs dictionary learning only on a small weighted randomized subset of the original samples (i.e., a coreset) instead of on all the input samples, making learning possible for massive data [GIM12]. Since constant blocks can be trivially encoded due to $\boldsymbol{d}_1$, we estimate for each input block the potential residual error $e_i = \|\boldsymbol{y}_i - (\boldsymbol{y}_i \cdot \boldsymbol{d}_1)\boldsymbol{d}_1\|_2^2$. In order to build a coreset of size C, we pick training samples with a probability proportional to e_i using a one-pass streaming method based on weighted reservoir sampling [Efr15], assigning a weight proportional to the reciprocal of the picking probability to account for the non-uniform input sampling. In contrast to COVRA [GIM12], we extract the coreset at negligible cost during the bottom-up construction of the downsampled pyramid and the near-lossless representation.

4.2.2 Elastic sparse coding

Given the optimal dictionary $\boldsymbol{D}$ for a given frame, we proceed to compute the optimal approximation that fits within our fixed-size pages. To do so, we encode pages in the order dictated by our HTA layout during a single streaming pass of the input volume. Our goal

Algorithm 1: Elastic OMP. *Algorithm for optimal allocation of sparse codes into fixed size pages*

Data: Dictionary $\boldsymbol{D}$, input blocks $\boldsymbol{y}_i$, page size P, brick size B, block size M, average non-zero count K
Result: For all i, encoding $I_i, \boldsymbol{\gamma}_i$ such that $\boldsymbol{y}_i \approx \sum_k \gamma_{ik} \boldsymbol{d}_{(I_i)_k}$ and $\sum_i count(I_i) = (PB/M)^3 K$

```
//Initialize
K_P = (PB/M)^3 K //Number of non-zeros in page
G = D^T D //Precomputed Gram matrix
//Compute solution for sparsity 0
foreach block i in 1..(PB)^3 do
    p_i = D^T y_i ; L_i = [1]
    //Compute null solution
    I_i = {} ; γ_i = {} ; α_i = p_i
    //Compute next best grow direction
    k_i^+ = {arg max_k ||α_ik||}
    γ_i^+ = {p_{i k_i^+}} ; α_i^+ = p_i − G_{k_i^+} γ_i^+ ; δ_i^+ = γ_i^+T (p_i − α_i^+)
    if δ_i^+ > 0 then push pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) into priority queue Q
end
//Greedy grow until page filled
while Σ_i count(I_i) < K_P and Q is not empty do
    //Select block with largest decrease in error
    pop pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) from priority queue Q
    //Move to next best solution for the block
    I_i = {I_i ∪ k_i^+} ; γ_i = γ_i^+ ; α_i = α_i^+
    //Compute next best grow direction
    k_i^+ = {arg max_k ||α_ik||}
    w = solve for w { L_i w = G_{I_i} } //Update Choleski decomposition
    L_i = [ L_i   0 ; w^T   sqrt(1 − w^T w) ]
    γ_i^+ = solve for c { L_i L_i^T c = α_i } ; α_i^+ = p_i − G_{k_i^+} γ_i^+ ; δ_i^+ = γ_i^+T (p_i − α_i^+)
    if δ_i^+ > 0 then push pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) into priority queue Q
end
```

is to find for all blocks i in a given page composed of P^3 compressed bricks composed of $(B/M)^3$ blocks, the best encoding $I_i, \boldsymbol{\gamma}_i$ such that $\boldsymbol{y}_i \approx \sum_k \gamma_{ik} \boldsymbol{d}_{I_{ik}}$, while $\sum_i count(I_i) = (PB/M)^3 K$. The general problem of optimal allocation of representation size to blocks is an instance of the multiple choice knapsack problem (MCKP), which is known to be NP-hard. Many efficient heuristic solutions exist [SZ79], but their application would require the precomputation of all possible errors associated to each possible non-zero count of block i. Here, we profit from the fact that efficient incremental solutions exist for sparse-coding to propose a greedy approximation to the allocation problem.

Our approach is based on a generalization of the greedy batch-OMP algorithm [RZE08], which was introduced for coding of a large number of signals over the same dictionary. The greedy batch-OMP algorithm selects at each step the dictionary column with the highest correlation to the current residual, orthogonally projects the input signal to the span of the selected dictionary columns, and recomputes the residual before repeating the process if convergence is not reached. High performance is ensured by replacing the pseudo-inverse in the orthogonalization step of standard Orthogonal Matching Pursuit with a progressive Choleski update. We exploit this progressive updating approach to compute in a greedy fashion the optimal sparse representation fitting in a page (see Algorithm 1). First, we pre-compute the Gram matrix $\boldsymbol{G} = \boldsymbol{D}^\mathsf{T}\boldsymbol{D}$ that is employed in the correlation search and orthogonalization steps. Then, for each of the blocks of the page, we compute the initial solution, using zero coefficients, and the associated residual. We also compute the next best solution by executing a step of the Batch-OMP method, and insert it into a priority queue sorted by maximum decrease in residual error. We then proceed by iteratively removing the top candidate from the queue, increasing its nonzero count by applying the precomputed solution, 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.

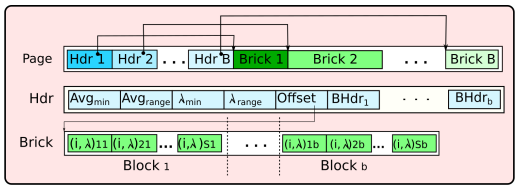

Figure 4: ***Lossy data page layout.*** *A constant-size page is composed by B brick headers and B brick data blocks. The headers (second row) are constant size and point to the associated variable-sized data block.*

When the greedy algorithm terminates, we store the solution in a compact constant-size page format, depicted in Fig. 4. Each block is approximated using an average value plus a variable number of (index, value) pairs for the sparse representation. The first bytes of the page representation are dedicated to B constant-size brick headers, which point to the variable-size brick content. The brick layout uses 32-bit aligned words and is designed for maximum decoding performance, rather than maximum compression. In particular, we do not apply any bit transformation and entropy coding. The brick header contains 4 32-bit floats for dequantization (range of average and range of coefficients), a 32-bit offset to the block representations, and b 32-bit block descriptors. Each descriptor contains the 8-bit nonzero count for the block, a 12-bit quantized version of the block average, and a two 6-bit quantized versions of the range of the coefficients of the block. The range is expressed with respect to the coefficient range of the entire brick. The rest of the page is dedicated to the sparse-coded representation, which concatenates for each brick the (k,γ) pairs using 16 bits/pair, with $bitcount(k) = \log_2(n)$ and $bitcount(\gamma) = 16 - bitcount(k)$. Using such alignments permits the decoding kernel (see Sec. 5.2) to read the data representation with 32-bit aligned fetches, without introducing bank conflicts.

The approach proves very appropriate for low-bitrate encoding in our context, as it provides state-of-the-art quality at below 0.5 bps for floating point data, while ensuring fast random-access decoding (see Sec. 5). In particular, the decoding performance on GPU is sufficiently high ($\gg$10 GVox/s on a NVIDIA GeForce GTX 1080Ti) to make it possible to use this encoding for temporal exploration of massive dynamic datasets. See Sec. 7 for more details.

5 Adaptive rendering from compressed data

Our GPU-accelerated rendering architecture exploits the fact that single frames can be fully stored in graphics memory in the HTA format using the GPU-friendly low-bitrate representation. This permits us to implement the high-bandwidth rendering process by mirroring the out-of-core representation to graphics memory through a paging process, and to implement transient decompression and rendering as GPU operations working on resident structures.

5.1 GPU-accelerated configurable renderer

The process of rendering a single timestep is described in Algorithm 2, and is made efficient by exploiting a number of data structures visible to the CUDA kernels (see Fig. 5).

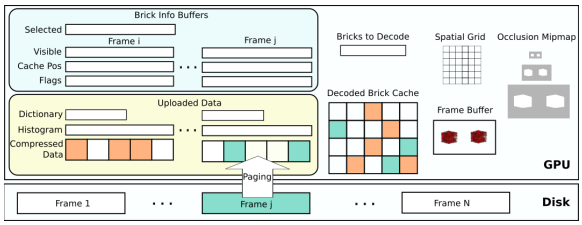

Figure 5: ***GPU data structures.*** *We support rendering several timesteps in a single frame. On board, we perform rendering by accessing bricks stored in the HTA structure. The low-bandwidth representation is mapped to graphics memory through a paging process. A decoded brick cache as well as other transient structures support ray-guided streaming and rendering.*

In order to support rendering of multiple timesteps at once (see Sec. 6), we preallocate at startup space for the maximum number of time steps that can be rendered together. At each frame, given the selected time step t to be visualized, the histograms and the dictionary are uploaded if not already present (line 1). The information required for forward and backward index computation (Sec. 4) is instead loaded once per dataset, as it is constant for all frames. Then, in parallel for each candidate brick, given the viewing parameters V, the current transfer function τ, and, if animation is not currently active, visibility feedback from previous frame, we determine the set of bricks that form the current potentially visible variable resolution representation of the dataset (line 2). At the same time, the selected brick indexes are inserted in a spatial index grid, which is a regular grid with the same spatial extent of the volume and cells with the size of the bricks in the finest hierarchical level. We then sort the indices of the potentially visible bricks into a number of slabs orthogonal the axis most aligned with the viewing direction (line 4). The number of slabs depends on the situation, since they are used to enable rendering of frames too big to be fully decoded at once on the GPU, as well as to reduce useless decoding by exploiting occlusion culling. If we are exploring a single frame and the full working set of the current frame can fit in cache, the number of slabs is set to 1 to perform single-pass rendering, otherwise it is set by default to a fixed small number (2 for static data benefiting from multi-frame caching, 8 for dynamic data that refresh the cache at each frame), but slabs that contain more than the number of bricks than can be decoded at once and stored on the GPU are further subdivided. The rendering process then proceeds on a slab-by slab basis (lines 5-14), traversing them in front-to-back order. First, we further reduce the working set by removing from the potentially visible set the bricks that are occluded by data rendered in a previous slab (line 6). This conservative culling process, active also for dynamic datasets, is supported by a mipmapped occlusion buffer, which maintains the achieved opacity for each pixels. Bricks are considered occluded if their projection is covered by full-opacity pixels. Bricks which survive are marked as needed for rendering the current slab (line 7). Since out-of-core bricks do not overlap, we expand the working set with the bricks containing the extra 2 voxels required for trilinear interpolation and gradient computation.

Algorithm 2: Rendering process. *Structure of the GPU-accelerated rendering algorithm from out-of-core data.*

Data: timestep t, view parameters V, transfer function τ, screen-space tolerance ε
Result: updated framebuffer, occlusion info for ray-guided streaming, and GPU caches

```
1  init(t)
2  identify_renderable_sets(V,τ, ε)
3  cleanup_cache()
4  sort_renderable_set_by_slab(V)
5  foreach slab in front to back order do
6      cull_occluded_bricks()
7      identify_bricks_to_decode()
8      upload_missing_pages()
9      associate_bricks_to_cache_positions()
10     decode_bricks()
11     fill_brick_aprons()
12     raycast(V,τ)
13     update_occlusion_mipmap()
14 end
15 pullup_visibility()
```

A decoded bricks cache supports the rendering task: if we are dealing with static data, decoded bricks are cached for reusing over multiple frames, otherwise the cache is cleaned up at each new frame. If we are rendering from the low-bitrate representation, we first page in all the pages that contain needed bricks that are not in the decoded brick cache (line 8). For the near-lossless representation, this is not needed since we do not incrementally upload and maintain the compressed representation in graphics memory. In both cases, the decoded brick cache is then updated. Bricks in the cache are computed to be self-sufficient during rendering, and, thus must have a 2-voxel apron, i.e., a border around each brick that duplicates the values across brick boundaries in order to permit to exploit texturing operations for trilinear filtering and gradient computations. Cache updating is thus performed in two passes, first decoding bricks from a compressed representation into the inner voxels of bricks in the decoded cache (line 9-10), and then filling-in the apron voxels by copying data from the already decoded neighboring bricks (line 11). Avoiding to store aprons of 32^3 bricks saves 42% of storage space and data transfer bandwidth.

With all the required data in cache, rendering is then performed by raycasting (Line 12), following per-pixel rays limited to the current slab extent. While all previous operations where parallelized on a brick-by-brick basis, this step uses a GPU thread per pixel. Ray traversal exploits the dynamically computed spatial index for brick identification and empty space skipping, and supports early ray termination by stopping when maximum opacity is reached and updating the occlusion mipmap (updated in line 13 and used in line 6). In addition, in order to support ray-guided streaming, the visibility feedback status of each traversed brick is set to true.

After rendering all slabs, the frame-buffer contains the final composited image for the volume, and the visibility status of all rendered bricks is up-to-date and can be used to guide the next frame's refinement step (line 2). Since the visibility status is computed only for bricks which are leaf in the current representation, at frame end it is pulled up to coarser grid levels by considering visible a brick with at least one of its finer level bricks was traversed (15).

5.2 GPU-accelerated operations

The rendering scheme can be fully implemented using a series of well-defined GPU-accelerated operations. The operations done at initialization stage are the following.

- **init()** first verifies if the current timestep is in the active set, and, if not, uploads to graphics memory its histogram and dictionary for low-bitrate decoding. If the active set is full, and all active timesteps are for the current frame, the most recently used one is replaced. Otherwise, the oldest one is replaced. We then clear the viewport to the background color, the occlusion mipmap to null opacity, and the visited state of bricks to false.
- **identify_renderable_sets()** is executed as a single GPU kernel,

with one thread per brick. Since our mapping allows us to derive, from the brick position, its level and 3D location, all bricks can be handled in parallel. The bricks are thus marked part of the potentially visible set if within the view-frustum, having some voxels at non-null opacity according to the precomputed histogram, and projecting its voxel at the correct size according to the current screen-space tolerance. When inter-frame occlusion is enabled, we discard the brick if both the brick and its parent were not visible on previous frame according to the visibility buffer. If the brick is selected the kernel marks it as a leaf on the flag buffer and writes the slab index and the brick id on the selected bricks buffer, see Fig. 5. For efficient empty space skipping, all cells of the spatial index covered by the brick are initialized with the brick identification $(level, x, y, z)$ (4 x uint8), where *level* has the highest bit set to one if the brick is empty. At kernel termination, thus, the spatial index correctly indexes the multi-resolution representation used for the current frame, so that, during traversal, it is possible to enter a brick, access its data through the layout function, and proceed to the next brick using the extent dictated by the level.

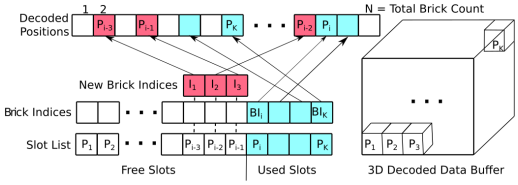

Figure 6: ***Cache buffers.*** *Slot list contains all the slots in the 3D buffer. The index of the corresponding brick is stored for each used slot (in cyan). The indices of the new bricks to be decoded are in magenta. Corresponding slots are copied from the slot list to the decoded positions buffer.*

- **cleanup_cache()** is executed as a GPU kernel, with one thread per used cache slots for usage identification, followed by a GPU compaction step. The cache, see Fig. 6, is organized in two parallel buffers, with the first containing all the cache brick positions and the second containing the associated compound id *(brick, time step slot)*. Associating the time-step slot to the id makes it possible to handle multiple timesteps at once. The first *available_slot_counter* slots are free, and the last are the used ones. This kernel sets to free all slots pointing to bricks not part of the current working set. A GPU compaction step then places the free slots before the used ones and updates *available_slot_counter*.
- **sort_renderable_set_by_slab()** performs GPU sorting of the buffer of selected bricks according to the slab id. After this operation all the selected bricks belonging to a slab can be easily accessed linearly through this buffer.

After these initial operations, we iterate the following sequence of steps for each slab.

- **cull_occluded_bricks()** is executed as a GPU kernel, with one thread per selected brick in the current slab. Exploiting the occlusion mipmap, it checks if the brick is occluded by identifying at which level of the mipmap the box projects to a single pixel and then checking if all the 2×2 pixels covering the box projection are conservatively marked as fully occluded. The brick flag buffer is updated accordingly.
- **identify_bricks_to_decode()** is executed as a GPU kernel, with one thread per selected brick in the current slab. It is responsible for identifying which are the bricks which need to be decoded, i.e., the currently unoccluded brick in the working set which are not already in cache. In this phase, we also identify the *auxiliary bricks* which are necessary to compute the apron voxels. Hence for each active brick, we check all its 26 neighbors and mark as *auxiliary* the ones not marked as active and not present in cache.

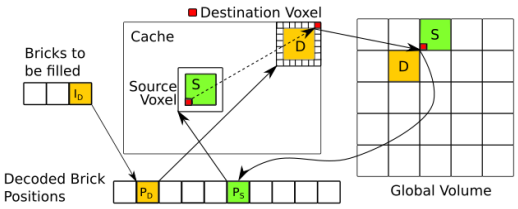

Figure 7: ***Apron Filler.*** *The apron voxels of destination yellow brick D are filled following these steps: (1) find cache slot of I_D; (2) from cache slot and voxel position, compute global $(level, x, y, z)$; (3) convert $(level, x, y, z)$ to an offset in the HTA layout to identify source brick (green); (4) find corresponding position in cache, fetch voxel and copy to destination position. Only one apron layer (instead of two) is depicted here for simplicity.*

- **upload_missing_pages()** is responsible of mirroring to graphics memory the pages of the low-bitrate representation which contain active bricks. This operation can be automatically performed using CUDA Unified Memory Access (UMA) [Nvi] by mapping the out-of-core file to graphics memory. We have however found that significantly higher performance can be obtained by implementing paging in an explicit memory manager, by updating a small buffer of flags (one byte/page) during brick identification, downloading it to CPU, and moving to GPU with `cudaMemcpyAsync()` the missing pages.
- **associate_bricks_to_cache_positions()** is executed as a GPU kernel, with one thread per brick to be decoded. Each thread associates to each brick the available position in the slot list associated to its thread id. The cache positions are copied to the decoded position buffer, while the brick indices are written in the cache index buffer. Finally, *available_slot_counter* is updated.
- **decode_bricks()** computes the decompressed brick representation, and behaves differently if rendering static frames from the near-lossless representation or high-frequency updates from the low-bandwidth representation. In all cases, only the inner voxels of the bricks are computed, since apron voxels will be separately infilled later on. The decoding of each brick is thus totally local. Near lossless data just uploads to the cache the decompressed version of each brick, exploiting the ZFP codec [Lin]. Decompression from the low-bitrate representation starts by computing, with a GPU kernel, for each brick, the offset into the compressed data representation of each variable-rate block and the dequantized version of the block average and coefficient ranges. Decompression is then started, using a thread per decoded voxel, with a grid size equal to the block size. The decoding threads cooperatively upload the compressed block representation to shared memory. Each participating thread loads 32 bits, and decodes to GPU shared memory two index-coefficient pairs in order to avoid bank conflicts. After thread synchronization, each thread separately computes the linear combination of its associated elements required to compute its associated voxel.
- **fill_brick_aprons()** computes the 2-voxel apron of each newly decoded brick that is not flagged as auxiliary. The operation is

performed by three GPU kernels: one for the top+bottom layers, one for left+right, and one for back+front of each brick. We've experimentally found that decoding 4 voxels per thread is a good trade-off between computation and read/write operations. Threads are associated to the boundary voxels of destination bricks. For each of these voxels, the original source brick and the associated voxel is identified by exploiting the decoded position buffer, which maps global positions to decoded bricks, and the known correspondence between local coordinates in the source brick and destination brick (Fig. 7). This procedure is applied once per slab for newly decoded bricks missing their apron.

- **raycast()** performs the rendering and accumulation of the portion of the rays in the current slab by accessing the decoded information through the spatial grid. The operation is performed using one ray per thread, which traverses the regular grid using a DDA approach. Using the spatial grid results in a faster approach with respect to KD-restart [FS05], or neighbor pointers [GMI08], because it permits us to skip any further hierarchy descents, which are generally used to traverse other similar hierarchical volume structures. Each time a grid cell is entered, the type and location and size of the indexed brick are computed from the stored $(level, x, y, z)$ of the cell. Empty bricks are accumulated at once, while non-empty bricks are accessed by fetching their cache position from the decoded brick position buffer before performing accumulation of all the voxels. Then accumulation continues by moving to the exit cell, determined by the ray direction and the brick size. The process repeats until the ray terminates because of full opacity or the end of the slab is reached. When a brick produces a not empty contribution, a visibility flag is set to one in the visibility buffer. Writing can be done simultaneously without the need of atomic functions, because all writes of concurrent threads will be for the same value and there are no race conditions. Moreover, when full occlusion is reached, the occlusion value of the pixel is updated.
- **update_occlusion_mipmap()** builds the mipmap of occlusion values from the fine-level occlusion map, in order to permit an efficient intra-frame occlusion computation. A kernel with one thread per pixel of the coarser map version is executed for each mipmap level. Intra-frame occlusion culling is fully conservative, and it is used to reduce the number of decoded bricks.
- **pullup_visibility()** updates the inner node visibility values from the leaf-level visited-node values, in order to permit an efficient inter-frame occlusion computation for ray-guided streaming. This operation is performed bottom-up in CPU by marking as invisible all nodes below the currently rendered representation, and as visible each node for which at least one child is visible. Inter-frame occlusion culling is used to reduce data uploading and decoding through ray-guided streaming. Since this non-conservative operation may lead to unwanted dynamic effects due to incremental loading during animations, it is only activated for rendering static frames.

6 Data distribution and rendering clients

Our flexible rendering process makes it possible to generate a variety of systems exploiting both highly-compressed rate and high-quality data. The default configuration, depicted in Fig. 2, supports both thin clients receiving all rendered images from server-side renderers, and fat clients performing local high-frequency rendering on a locally-downloaded copy of low-bitrate data and using a separate process (local or on a remote rendering server) for generating high-quality snapshots from near-lossless data.

In our reference implementation, a user interface freely moves the camera, changes the transfer function, and determines which timesteps are shown. Available controls include using a jog shuttle to jump to a selected timestep or to interactively move forward and backward in time, as well as enabling playback with user-defined speed and time direction. Moreover, all the clients support rendering images associated to a single simulation timestep, as well as rendering multiple timesteps in parallel viewports, maintaining the same transfer function, viewing parameters, and animation settings. This latter option is made possible by the fact that we cache on GPU multiple frame descriptions (Sec. 5.1). This option makes it straightforward to render parallel evolution in separate viewports (see Fig. 8 and accompanying video), but other more advanced options for presenting multiple frames are also readily implementable [HMCM09, BCP*12]. In contrast to the approach of Pulido et al. [PLK*18], which also used multiple frames displayed at once to improve temporal understanding of simulations, we are not limited to multiple static frames, but we can perform spatiotemporal exploration on the selected frames, obtained by keeping the time difference between displayed frames constant while we move in time and by sharing the same interactive camera.

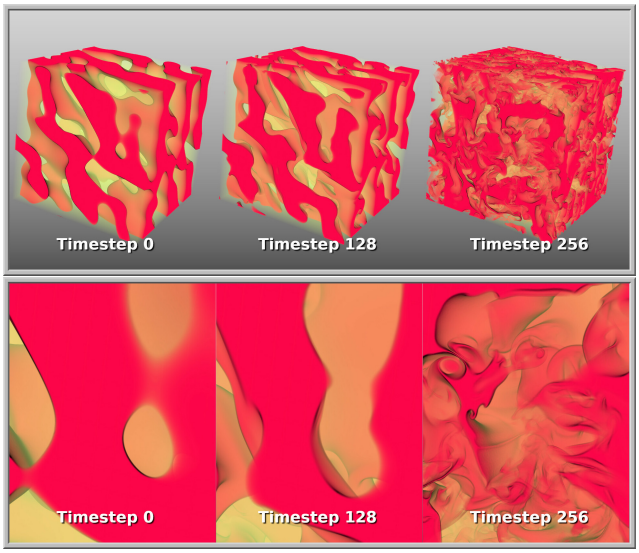

Figure 8: ***Temporal multiview rendering.*** *Three time steps of the HBDT dataset synchronously inspected within the same frame.*

Fat client and snapshot renderer. The fat client (Fig. 1 left) begins its operations by progressively downloading the low-bitrate representation of the explored dataset. The default progressive downloading behavior is to first download one every eight timesteps, and for each timestep download the data at half resolution, leading to a 1/64 data reduction. After this initial data arrives, interaction can begin, and data is continued to be downloaded in background, first refining along the temporal direction, and, then refining in space. Implementing this progressive method is straightforward, since timesteps are stored separately and each timestep is stored in coarse-to-fine order. For a 1024×1024^3 dataset, we thus need to receive only about 1 GB of data before starting to interact, and 64GB to receive

the full dataset. On a typical 1 Gbit/s network this translates to an initial latency of only a few seconds, after which the user can start to perform the first inspection operations (e.g., browsing the evolution, selecting transfer functions) while waiting for the full data. During interactive exploration, rendering operations are controlled by tuning the renderer behavior according to the interaction state. We use a simple state machine for implementing this behavior. When continuously moving among timesteps (playback, jog&shuttle), the renderer is configured to favor time continuity over spatial continuity. In this case, inter-frame occlusion is disabled. By contrast, when animation is stopped, and the user is moving, inter-frame occlusion is enabled. As soon as the user stops moving or changing other view parameters (including transfer function) for more than one frame, an asynchronous request is sent to the snapshot renderer (typically residing on a server), which immediately starts producing a high-quality frame by using the rendering algorithm with the near-lossless data. The rendering process is carried out on a slab-by-slab basis and is interruptible. When the client moves, it can thus immediately cancel the request if the high-quality image has not yet arrived. For end-to-end bandwidth reduction, we use a CUDA JPEG encoder to compress the framebuffer on GPU before downloading it for network transmission [HSP*13]. At maximum quality, we can encode and transmit 4K images at <200ms latency on a NVIDIA GTX 1080Ti, which is sufficient for static snapshots.

Thin client and remote interactive renderer. The thin client design is meant to work on platforms not capable of high-frequency volume rendering (Fig. 1 right). The typical use case is mobile rendering. In order to show the feasibility of the approach, we implemented an Android client with the same user interface and behavior as the fat client. The only major difference is that high-frequency frames are delegated to a remote renderer, sent over the network as images, and displayed locally. The remote renderer starts as a streamlined copy of the fat client without a user interface, and begins by waiting for client connection. Then, it continuously uses the view and transfer function parameters communicated by the client to generates images, encode them and send them for display to the client. The frame grabber and encoder used in this work is the same CUDA codec used for high-quality images, configured for high compression rate (for this paper, quality 75%, no chroma subsampling). An alternative, and more promising, solution would be to use the H.265 hardware decoder of NVIDIA boards. We are currently not using it as, in our first implementation, we had an increased latency on Android clients. We plan to investigate these issues in the future. With our current codec, a full HD image is encoded to ~0.5MB/frame in less than 8ms, which is largely compatible with interactive speeds on a typical network setting. At this compression rate, 20 frames/s translate to a 80 Mbit/s stream, which is fully within typical wireless limits, and will be readily usable also in broadband settings, given the current evolution towards 5G networks, with a predicted 490 Mbit/s median speed for consumer-level 3.5GHz bands.

7 Implementation and results

An implementation of our architecture has been realized with C++, OpenGL, and NVIDIA CUDA 10.0, on Linux for processing, server-side rendering, and fat clients, and on Android for the proof-of-concept thin client. We have tested it with a variety of high resolution static and dynamic models. In this paper, we discuss the results obtained with three representative massive time-varying datasets from the JHU Turbulence database [Tur]: the velocity magnitude field of a forced isotropic turbulence simulation (*ISO*, 1024 timesteps 1024^3, float, 4TB), the density field of a homogeneous buoyancy-driven turbulence simulation (*HBDT*, 1010 timesteps 1024^3, float, 4TB), and the x-component of the velocity field of a channel flow simulation (*CHAN*, 4000 timesteps $2048 \times 512 \times 1536$, float, 24TB). In addition, to test rendering performance also on massive static dataset, we present rendering results for a single frame of the density field of a Rayleigh-Taylor instability simulation [LBM*06] (*RT*, 1 timestep 3072^3, ushort, 54 GB). Fig. 9 present several frames of inspection sequences of these datasets.

7.1 Compression performance

We evaluated our low-bitrate compression strategy by running a battery of tests on the selected time-varying datasets, changing the target non-zero count K to $6, 9, 12, 15$ to obtain compression rates between ~0.25 bps and ~0.50 bps. In all tests, we used pages of $P^3 = 4^3$ bricks of $B^3 = 4^3$ blocks of $M^3 = 8^3$ voxels. We learned dictionaries of 1024 prototype blocks in 50 K-SVD iterations on 16 Mvoxels coresets. While a number of possible settings are supported in our system, the above configuration is tuned for low-bitrate encodings. In particular, using power-of-two settings aligns well with our multiresolution grids that halve resolution at each coarser level. In this context, the selected block dimension is the smallest that can support the required compression rate, while 1024 prototypes can be indexed with 10bits, leaving 6 bits for coefficient encoding, and are sufficient to generate an overcomplete dictionary for the selected block dimension. The number of iterations and coreset size were selected based on prior experience with similar encoders [GIM12].

In order to provide a context for the evaluation of the compression component of our work, we provide also a comparison with alternative compression methods supporting real-time decoding (i.e., over 1 GVox/frame at 10 frames/s). In particular, we evaluate a fixed-rate encoding (similar to COVRA [GIM12]), ASTC [NLP*12] (version 10/2017 [ARM]), and Hierarchical Vector Quantization (HVQ) [SW03] (version used for the COVRA evaluation [GIM12]). We also include, as a reference, a comparison with methods that do not meet the decoding speed constraints, i.e., the CudaCompress wavelet codec (CC) [TBR*12] (version 2013 [Tre]), ZFP [Lin] (version 0.5.4 [Lin]), and SZ [DC16] (version 2.1.0 [ANL]). 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, while for CC we prescribed the required bit rate, for SZ we varied relative error, and for ASTC we selected maximum quality and lowest bitrate achievable. Using settings similar to the COVRA evaluation [GIM12], HVQ was run with a dictionary of 1024 elements per level and 12 bits for the quantization of block average.

Compression Speed. Processing was performed on a single Arch Linux PC with 256MB RAM and two 24-core Intel Xeon E5-2650 v4 2.20 GHz CPUs, with input and output data stored on a Synology RS3617Xs+ with a RAID 5 composed of SEAGATE ST10kNE004-1ZF101 disks (BTRFS file system) connected using a 10 Gbit/s link. Compression was performed using four parallel processes, encoding four frames at a time. The analysis of the different phases of

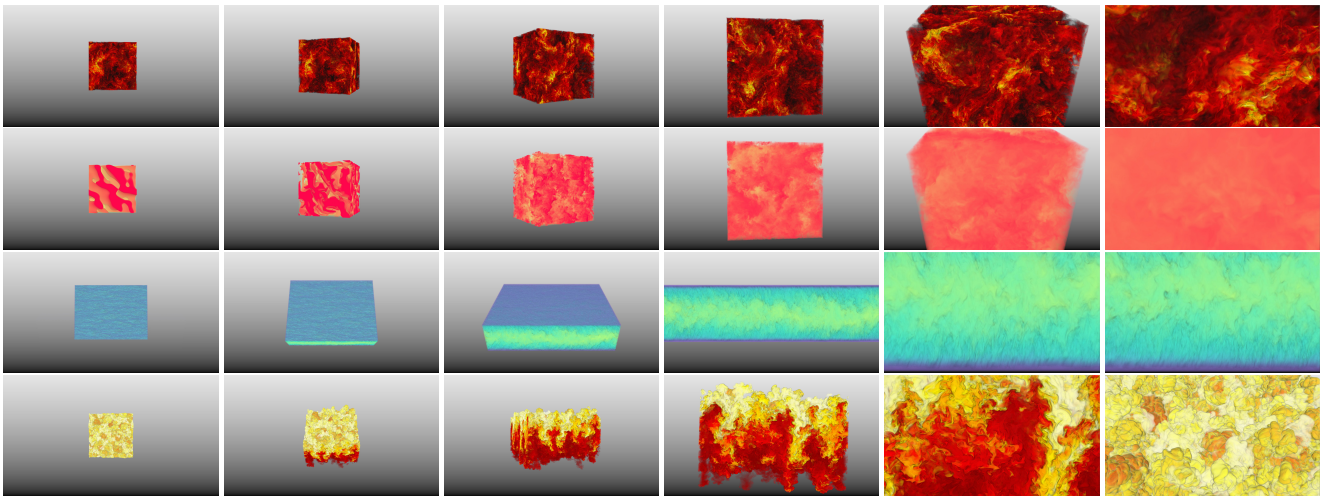

Figure 9: ***Inspection sequences.*** *Representative frames of interactive spatiotemporal inspection sequences of massive dynamic and static datasets. From top to bottom: ISO (*1024 *timesteps* 1024^3*), HBDT (*1010 *timesteps* 1024^3*), CHAN (*4000 *timesteps* $2048 \times 512 \times 1536$*), RT (*1 *timestep* 3072^3*)*

construction shows that the bottom-up filtering phase (eventually including coreset extraction phase), common to all methods is reasonably fast (20 s/frame for *ISO* and *HBDT*, 32 s/frame for *CHAN*), and is dominated by data transfer time from the file server to the processing node. The various tested codecs have very different coding times. The faster is the hardware-accelerated CC, which is able to encode data at a very high rate (about 4 s/frame for *ISO* and *HBDT* and 6 s/frame for *CHAN*), followed by ZFP (15-29 s/frame for *ISO* and *HBDT*, 23-44 for *CHAN*, lower times being for higher compression rates). The slowest codecs are ASTC (over 12 min/frame for *ISO*, 15 min/frame for *HBDT*, 17 min/frame for *CHAN*) and SZ (13 min/frame for *ISO* and *HBDT*, 21 min/frame for *CHAN*). The compression times for the sparse coding methods fall somewhere in the middle between these extremes. Dictionary learning time is independent from dataset size and only slightly dependent on sparsity (42-74s for $K = 6..15$). Encoding time is linear with dataset size, and grows sublinearly with target sparsity. Elastic encoding is, on average, between 1.5x to 2x slower than fixed-size encoding (24s-33 s/frame for fixed-size encoding vs. of 42s-74s for elastic optimization of *ISO* and *HBDT*, 35s-49s vs. 56s-74s for *CHAN*, lower times being for higher compression rates). While we do not consider encoding speed essential for this asymmetric application, where we just need to encode data once per simulation, we plan to improve compression speed on time-varying data by reusing dictionaries for neighboring frames and running only 1-2 refinement operations on these already trained dictionaries, since we expect dictionaries of neighboring frames to be similar. Given the amount of time dedicated to dictionary learning in the used configuration, we expect at least a 2x speed-up for this reusing strategy.

Decompression Speed. Supporting at least 10 frames/s animation requires a codec able to upload data to GPU, decompress, and render it at a rate of least 10 Gvox/s. Table 1 summarizes the uploading and decompression performance obtained on a single PC with i9-7900X 3.30 GHz CPU and a GeForce GTX 1080Ti for the various codecs. From the table, it is clear that, despite the increase in bandwidth, the uncompressed format is still not viable (less than 2.5 GVox/s from RAM to texture memory). The most performing real-time codec is ASTC, because of the direct hardware support. ASTC, however, has

	Real-Time								Non-Real-Time					
	Elastic		Fixed		ASTC		HVQ		Unc.		CC		ZFP		SZ	
bps	D	H	D	H	D	H	D	H	D	H	D	H	D	H	D	H
~0.26	33.33	29.41	41.67	35.71	–	–	–	–	–	–	4.18	4.11	0.23	0.14	0.01	0.01
~0.35	30.30	26.32	34.48	29.41	–	–	–	–	–	–	4.05	3.97	0.22	0.14	0.01	0.01
~0.45	27.03	23.26	31.25	26.32	–	–	–	–	–	–	4.01	3.92	0.20	0.13	0.01	0.01
~0.54	24.39	20.83	28.57	23.81	–	125.00	34.48	28.57	–	–	3.92	3.82	0.18	0.12	0.01	0.01
32	–	–	–	–	–	–	–	–	–	0.40	–	–	–	–	–	–

Table 1: ***Decoding performance.*** *Uploading and decoding speed in GVox/s for the various codecs. Real-time codecs have to support over 1 GVox/frame at 10 frames/s. Column D reports pure decoding speed for data in device memory for GPU codecs and in RAM for CPU ones, while column H reports the speed of moving data from host to device and decompressing it.*

limitations in terms of achievable compression and quality (see below). The sparse-coding approaches and HVQ also clearly meet the decoding speed constraint. It is interesting to note that elastic sparse coding is not introducing a major decoding overhead over fixed-rate coding, since the decoding kernel is carefully designed for speed rather than for maximum achievable compression. It avoids memory conflicts through aligned read/write operations and, since all voxels in a block are decoded in parallel and have the same non-zero count, it has minimum divergence. The fastest non-real-time codec is, by far, the CUDA-accelerated CC wavelet codec, which, however, falls below the required real-time decoding threshold, since, to achieve its excellent quality of reconstruction at low bitrates it includes several costly operations, such as inverse RLE decoding and Huffman decoding in addition to inverse wavelet transform. Moreover, for consistency with the original paper [TBR*12], we report here for CC the speed obtained for at a granularity of 256^3 voxels/brick, which produces the maximum achievable speed, while the other real-time codecs have been configured at a much finer granularity of 32^3, in order to support a more efficient culling. The CPU codecs ZFP, and even more SZ, cannot currently be used in a high-bandwidth context, both because of their limited speed and because data must travel in decompressed format from RAM to GPU. ZFP has very recently introduced a GPU-accelerated decoder (version 0.5.4 [Lin]), which, however, only supports fixed-rate encoding, with reduced quality over the fixed-tolerance version.

Rate and distortion. Table 2 summarizes the low-bitrate compression performance on a single selected time step of each time-varying

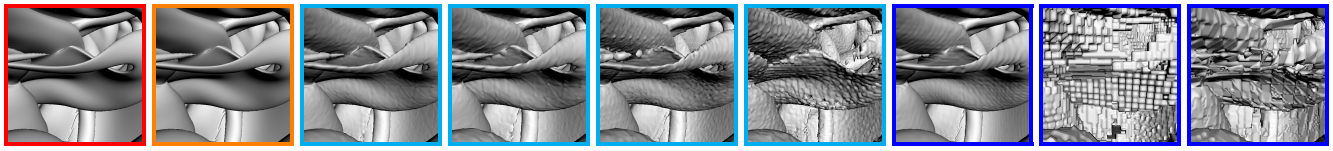

Figure 10: ***HBDT isosurface rendering quality.*** *From left to right: uncompressed, ZFP near-lossless (7.4 bps, volume PSNR=92.06, SSIM=0.999), elastic sparse coding (0.45 bps, SSIM=0.856), fixed sparse coding (0.43 bps, SSIM=0.809), ASTC (0.59 bps, SSIM=0.697), HVQ (0.50 bps, SSIM=0.351), CC (0.45 bps, 0.833), ZFP (0.41 bps, SSIM=0.206), SZ (0.46 bps, SSIM=0.347). Codecs supporting real-time decoding are marked in cyan, while others are marked in blue.*

	Real-Time								Non-Real-Time					
	Elastic		Fixed		ASTC		HVQ		CC		ZFP		SZ	
	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR
ISO	0.26	43.84	0.24	42.88	–	–	–	–	0.26	47.70	0.29	29.89	0.26	35.54
	0.35	45.80	0.34	44.86	–	–	–	–	0.35	49.71	0.34	33.07	0.34	35.73
	0.45	47.18	0.43	46.26	–	–	–	–	0.45	51.51	0.41	36.60	0.46	35.99
	0.54	48.22	0.52	47.34	0.59	45.85	0.50	41.47	0.54	52.89	0.53	40.52	0.55	36.28
HBDT	0.26	37.90	0.24	35.43	–	–	–	–	0.24	41.90	0.30	3.94	0.29	45.41
	0.35	39.61	0.34	37.40	–	–	–	–	0.32	44.64	0.36	13.54	0.33	45.56
	0.45	40.95	0.43	38.92	–	–	–	–	–	–	0.44	22.27	0.45	46.20
	0.54	41.66	0.52	39.91	0.59	38.02	0.50	23.28	–	–	0.58	30.44	0.55	46.88
CHAN	0.26	46.47	0.24	45.09	–	–	–	–	0.26	49.88	0.27	23.82	0.25	36.87
	0.35	48.43	0.35	47.10	–	–	–	–	0.35	51.98	0.34	33.21	0.33	37.07
	0.45	49.82	0.43	48.55	–	–	–	–	0.44	53.94	0.49	40.77	0.43	37.33
	0.54	50.80	0.52	49.64	0.59	23.17	0.50	41.70	0.52	55.47	0.63	44.81	0.51	37.65

Table 2: ***Compression rate and distortion.*** *The considered real-time methods are those capable to sustain a decompression rate of over 1 GVox/frame at 10 frames/s: elastic sparse coding and fixed sparse coding with varying sparsity (K=6,9,12,15), ASTC at highest quality and maximum compression, HVQ with fixed 1K dictionary. For reference, we also include results obtained with non-real-time methods: CC, ZFP with varying accuracy (-a), and SZ with varying relative error (RE). Empty cells correspond to non-achievable compression rate with the given method.*

dataset (time step 256 for all datasets). Other time steps provide consistent results. Empty cells correspond to non-achievable compression rate with the given method. To permit comparison 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 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, and y_i is the approximated one, and N the total number of voxels. Fig. 10 shows the effect of compression on image quality for an isosurface rendering detail of the HBDT dataset, and includes also the Structural Similarity (SSIM) [WBSS04] index of each image computed with compressed data with respect to ground truth. Based on decoding performance evaluation, the considered real-time methods are elastic sparse coding and fixed sparse coding with varying sparsity (K=6,9,12,15), ASTC at highest quality and maximum compression, and HVQ with fixed 1K dictionary. The scalability of our method is demonstrated by the fact that, by suitably tuning target sparsity, both fixed and elastic encoding can span a good range of both compression rates and quality. Elastic encoding proves to be able to significantly improve quality for all datasets in the tested compression range, as it increases PSNR by +1dB to +2dB over the fixed-size version, and by several dBs over the competing real-time methods. The improved compression quality of our method with respect to other real-time codecs also translates to significantly improved image quality at comparable bitrates (See Fig. 10). For reference, we have also included results obtained with non-real-time methods: CC, ZFP with varying accuracy (-a), and SZ with varying relative error (RE). CC proves to be the most performing codec at these low bitrates, with a performance that is slightly lower in terms of SSIM and superior in terms of PSNR with respect to elastic sparse coding, thanks to the implemented transformation and entropy coding methods, not included in the sparse-coding techniques to support real-time decoding. ZFP and SZ, even though they do not perform well at such a low bitrate, are more scalable than CC, which, by design, is not applicable in near-lossless mode, when many wavelet coefficients are non-zero and their quantization leads to large Huffman tables with many different symbols. The two empty cells in the table corresponds to cases where CC cannot achieve the desired bit rate, due to overflowing Huffman table. For this reason, we currently employ ZFP for near-lossless frames (see Diffenderfer et al. [DFH*19] for a study of the behavior of ZFP at moderate-to-high bitrates). Fig. 10 also includes the image generated by ZFP at 7.4bps (volumetric PSNR=92.06).

7.2 Rendering performance

The performance of our rendering system implemented as a fat client was evaluated on a single Arch Linux PC with 128MB RAM, a 24-core i9-7900X 3.30 GHz CPU, a GeForce GTX 1080Ti and a local Samsung 9160 Pro 1TB SSD for storage. The server is the same machine used for processing, connected on local 10 Gbit/s LAN. The performance of our proof-of-concept thin client implementation was evaluated, instead, on a Samsung Galaxy Note Pro SM-P905 Android 5.0 Tablet with a Qualcomm Snapdragon 800 chipset (Quad-core 2.3 GHz Krait 400 CPU and Adreno 330 GPU) connected at 144 Mbit/s on a moderately loaded wireless LAN to the same server used for the fat client. The remote renderer and the snapshot renderer were executed on the same node and shared all resources. The quantitative results presented here in details were collected by gathering information on pre-recorded interactive paths designed to be representative of typical volumetric inspection tasks and to heavily stress the system, including rotations and rapid changes from overall views to extreme close-ups and back (see also Fig. 9). Each recorded path was played back with different settings on a window size of 1920×1080 pixels. In order to stress the system, renderer resolution was always set to full accuracy (1 voxel/pixel). We measured actual frame rates (i.e., not only raw rendering times, but frame-to-frame times). The qualitative performance of our system is also illustrated in an accompanying video, that shows in live recordings of the analyzed sequences, as well as of similar interactive sequences with all datasets.

Startup latency. For these tests, we have used the 0.45 bps elastic setting, which leads to create low-bitrate representations of 65GB (ISO), 64GB (HBDT), 376GB (CHAN), and 1.8GB (RT). At start time, the system performs progressive downloading from remote storage, storing the representation on the local SSD disk for further access. Sufficient data for the initial exploration is received from the server after 2.5s for ISO and HBDT, 3.0s for CHAN, and less than one second for the single-frame RT. Almost immediate interaction is thus ensured. The full amount of data is received by the clients after less than two minutes for ISO and HBDT, about two minutes and a

half for CHAN, and less than three seconds for RT. This makes it viable to maintain data on a server, and download it on demand on local fast SSD disks only at the start of an inspection session. Using uncompressed data would require hours.

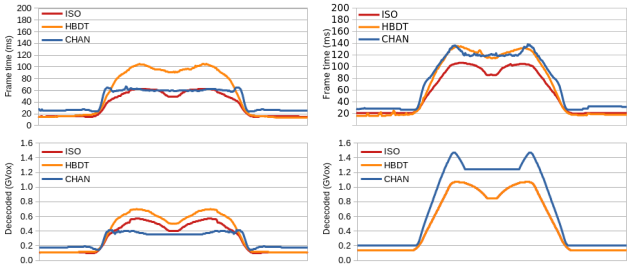

Figure 11: ***Dynamic dataset exploration.*** *Top Left: frame time with intra-frame occlusion culling. Top Right: frame time without intra-frame occlusion culling. Bottom Left: decoded voxels/frame with intra-frame occlusion culling. Bottom Right: decoded voxels/frame without intra-frame occlusion culling.*

Dynamic dataset exploration on fat client. When continuously moving among timesteps (playback, jog&shuttle), the renderer is configured to favor time continuity over spatial continuity. In this case, conservative intra-frame occlusion is enabled, while non-conservative inter-frame occlusion is disabled to avoid spurious dynamic effects. Fig. 11 reports rendering times and decoded voxel counts for the spatiotemporal paths in the accompanying video for the three dynamic datasets. Representative frames are in Fig. 9. In these spatiotemporal paths, the simulation is played first forward and than backward in time while the camera is moving. As reported, the frame rate is interactive with and without occlusion culling, since the average rendering time with occlusion culling enabled is 36 ms/frame for ISO 54 ms/frame for HBDT, and 44 ms/frame for CHAN, while the average frame times without occlusion culling are of 56 ms/frame for ISO 66 ms/frame for HBDT, and 73 ms/frame for CHAN. The peak rendering time when occlusion culling is enabled is for the highly transparent HBDT (105 ms/frame), while the peak when occlusion culling is disabled is for the larger, but more opaque, CHAN (138 ms/frame). Intra-frame occlusion is effective, at least for moderately opaque transfer functions, in improving performance by reducing the number of decoded bricks in conjunction with early ray termination. The total number of decoded bricks is reduced by 70% for ISO, over 148% for the more opaque CHAN, and 41% for the more transparent HBDT. This leads to rendering speedups of 55% for ISO, 88% for CHAN, and 22% for HBDT. Decoding speed is always maintained at around 19 GVox/s for all datasets, while apron filling has been measured at around 18 GVox/s. Since storing non-overlapping bricks reduces storage footprint, as well as server-to-client and client-to-GPU bandwidths by 42%, using the apron filling approach proves effective. For fully benchmarking random access capabilities, we also measured the performance obtained when playing back the same animation with randomly shuffled timesteps, which was in the range 18ms-122ms (average 42ms) for all frames and all datasets. This is just a bit below the achieved rendering speed, with an average performance decrease of 11% (at worst 81%), mainly due to the reduced efficiency in accessing the file system.

Static timestep exploration on fat client. When animation is stopped, and the user is moving, inter-frame occlusion is enabled,

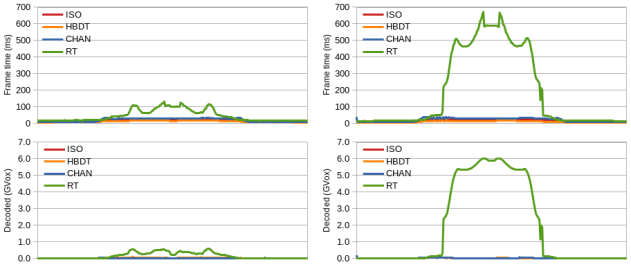

Figure 12: ***Static dataset exploration.*** *Top Left: frame time with occlusion culling. Top Right: frame time without occlusion culling. Bottom Left: decoded voxels/frame with occlusion culling. Bottom Right: decoded voxels/frame without occlusion culling.*

in order to increase spatial exploration performance even if incremental updates may introduce dynamic effects. In these conditions, the cache proves effective, and the number of decoded voxels/frames falls down due to ray-guided streaming Fig. 12 summarizes performance measures gathered during spatial exploration of the frame used for compression evaluation in Sec. 7.1. When the decompressed brick cache size is large enough to cache the entire working set, only few bricks/frame are decompressed at cache misses, and performance is similar to previous single-pass GPU raycasters [HBJP12, GIM12]. For the three dynamic datasets, using the same paths tested for the dynamic benchmark, the average frame rate is 54 frames/s for ISO and 78 frames/s for HBDT, and 45 frames/s for CHAN. The frame rate never falls below interactive performance, since the slowest measured refresh frequency is for CHAN at 27.1 frames/s. The average number of voxels decoded per frame never exceeds 106 Mvoxels, since much of the required working set for a given image is already present in cache before rendering. For these datasets, moreover, occlusion culling is less effective than for the dynamic case, since the loading and decoding effort per frame is already low due to caching. Thus, the frame rate with and without occlusion culling is almost the same. When, instead the dataset size is larger than the decompressed-brick cache size, as for RT, the situation changes. In this case, we switch to the multipass slab-based approach and we enable intra-frame occlusion culling in addition to the inter-frame one. At tolerance 1, we achieve for this dataset a minimum frame rate of 7.5 frames/s (average for path 22 frames/s), with a peak 599 decoded Mvoxels/frame. By contrast, without occlusion culling, the performance drops to a minimum frame rate of 1.5 frames/s, with a peak uploaded and decoded voxel count/frame of over 6 GVoxels/frame.

High quality still image on fat client. As soon as the user stops modifying the view for more than one frame, the client requests the remote snapshot renderer to produce a high-quality frame by applying our out-of-core raycaster to the near-lossless data. Using a tolerance of 1 voxel/pixel, the latency measured for receiving the image rendered from near-lossless data has been measured to 0.5-1.5s for the dynamic datasets, and to 0.8-7s for the much larger static RT dataset (timing depending on views). Almost all time is due to loading and decoding data from storage (fetching data from the file server rather than from local SSDs), and ZFP decoding and uploading of bricks. Even though these aspects can be optimized (e.g., by using a CUDA ZFP decoder or a faster file server), the latency is already acceptable for an interactive application, at least for our time-varying data.

Thin client performance. As rendering for a thin client is done remotely, the difference in performance is mostly due to the increased latency and reduced frame rate connected to the communication. We evaluated our proof-of-concept implementation by re-executing the dynamic tests on the Android client. We found that, on our current implementation, the remote renderer speed remains, within a few percents, exactly the same as that of the fat client (Fig. 11), since the only overhead is GPU encoding of rendered image and socket transmission. The limiting factor, at 1080p, is, on our implementation, the proof-of-concept Android client. A constant ~30 ms/frame is spent in network operation and OpenGL refresh, while the most demanding operation is the software decoding of the received images, which takes ~98 ms/frame. As a result the refresh speed is capped to ~8 frames/s on all datasets. Lowering resolution to 720p reduces decoding time to ~46 ms/frame increases maximum client frame rate to ~13 frames/s. We expect to remove this limitation by using an optimized decoder on Android, but this implementation issue is orthogonal to our work.

8 Conclusions

We have presented a novel flexible approach to support time-varying rectilinear scalar volume exploration. By introducing a novel high-performance low-bitrate codec, which advances the state-of-the-art in terms of quality achievable at very low bitrates in real-time settings, and combining it with a near-lossless codec within a new software architecture, we are capable of supporting full spatial and spatiotemporal exploration in a variety of setups, from local analysis on graphics workstations to remote exploration on thin mobile clients.

The major limitations, shared with other compression-based approaches, are the non-negligible encoding time and the (unavoidable) limits of the achievable quality during animation dictated by the need to fit massive frame sizes in the available bandwidth. Our results show, however, that our solution is of immediate practical interest, since excellent-quality results can be achieved on time-varying datasets with billions of voxels per frame and thousands of time-steps, and, by combining system-level solutions for spatial exploration and for automatic generation of full-quality static frames, many use cases can be handled.

Our future work will further extend the capabilities of this approach, in particular moving from scalar data to multidimensional data, where we expect that the benefits of our compression-based approach can provide further advantages. Moreover, while in this work we focused on the enabling technology to support a variety of interactive exploration data-intensive means, we plan in the future to complement these features with domain-independent and domain-dependent data-reduction methods for extracting relevant information from time-varying data, bridging the gap between interactive raw-data exploration and static derived-data display.

Acknowledgments. The authors would like to warmly thank Peter Lindstrom (ZFP), Marc Treib (CC), and Sheng Di, Dingwen Tao, Xin Liang (SZ) for making their compression codes available. Datasets ISO, HBDT and CHAN are courtesy of the Johns Hopkins Turbulence Database (JHTDB) initiative. Dataset RT is courtesy of LLNL. We also acknowledge the contribution of Sardinian Regional Authorities (projects VIGECLAB and TDM) and of King Abdullah University of Science and Technology (KAUST).

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. 3, 4

[ANL] SZ compression library. https://github.com/disheng222/SZ. [accessed: 2018-10-31]. 9

[ARM] ASTC compression library. https://github.com/ARM-software/astc-encoder. [accessed: 2018:10:31]. 9

[BCP*12] BRAMBILLA A., CARNECKY R., PEIKERT R., VIOLA I., HAUSER H.: Illustrative flow visualization: State of the art, trends and challenges. *Proc. EG STAR* (2012). 8

[BGH*06] BIKSHANDI G., GUO J., HOEFLINGER D., ALMASI G., FRAGUELA B. B., GARZARÁN M. J., PADUA D., VON PRAUN C.: Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. PPoPP* (2006), pp. 48–57. 3

[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, 4

[BRLP18] BALLESTER-RIPOLL R., LINDSTROM P., PAJAROLA R.: TTHRESH: Tensor compression for multidimensional visual data. *arXiv preprint arXiv:1806.05952* (2018). 2, 4

[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

[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. 2

[DC16] DI S., CAPPELLO F.: Fast error-bounded lossy HPC data compression with SZ. In *Proc. IEEE IPDPS* (2016), pp. 730–739. 9

[DFH*19] DIFFENDERFER J., FOX A., HITTINGER J., SANDERS G., LINDSTROM P.: Error analysis of ZFP compression for floating-point data. *SIAM Journal on Scientific Computing* (2019). To appear. 11

[Efr15] EFRAIMIDIS P. S.: Weighted random sampling over data streams. In *Algorithms, Probability, Networks, and Games*. 2015, pp. 183–195. 4

[Ela08] ELAD M.: *Sparse and Redundant Representations*. Springer, 2008. 2

[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

[FE17] FREY S., ERTL T.: Flow-based temporal selection for interactive volume visualization. *Computer Graphics Forum 36*, 8 (2017), 153–165. 1

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

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *Proc. Graphics hardware* (2005), pp. 15–22. 8

[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

[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, 9, 12

[GMI08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J. A.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7–9 (2008), 797–806. 8

[GS01] GUTHE S., STRASSER W.: Real-time decompression and visu-

alization 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, 12

[HMCM09] HSU W.-H., MEI J., CORREA C. D., MA K.-L.: Depicting time evolving flow with illustrative visualization techniques. In *International Conference on Arts and Technology* (2009), Springer, pp. 136–147. 8

[HSP*13] HOLUB P., SROM M., PULEC M., MATELA J., JIRMAN M.: GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video. *Future Generation Computer Systems 29*, 8 (2013), 1991–2006. 9

[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

[ILRS03] IBARRIA L., LINDSTROM P., ROSSIGNAC J., SZYMCZAK A.: Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum 22*, 3 (2003), 343–348. 2

[Iri06] IRION R.: The terascale supernova initiative: Modeling the first instance of a star's 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. 1, 2

[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. 2

[LBM*06] LANEY D., BREMER P.-T., MASCARENHAS A., MILLER P., PASCUCCI V.: Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE TVCG 12*, 5 (2006), 1053–1060. 9

[Lin] ZFP compression library. https://computation.llnl.gov/projects/floating-point-compression/zfp-versions. [accessed: 2018-10-31]. 4, 7, 9, 10

[Lin14] LINDSTROM: Fixed-rate compressed floating point arrays. *IEEE TVCG 20*, 12 (2014), 2674–2683. 2, 4

[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

[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. 2

[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. 2

[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. 2

[NJ16] NOGUERA J. M., JIMÉNEZ J. R.: Mobile volume rendering: past, present and future. *IEEE transactions on visualization and computer graphics 22*, 2 (2016), 1164–1178. 2

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *Proc. HPG* (2012), pp. 105–114. 2, 9

[Nvi] CUDA toolkit. https://developer.nvidia.com/cuda-toolkit. [accessed: 2018-10-31]. 7

[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, 8

[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

[RZE08] RUBINSTEIN R., ZIBULEVSKY M., ELAD M.: *Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit*. Tech. rep., CS Technion, 2008. 5

[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, 2

[She06] SHEN H.-W.: Visualization of large scale time-varying scientific data. *Journal of Physics 46*, 1 (2006), 535–544. 2

[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, 9

[SZ79] SINHA P., ZOLTNERS A. A.: The multiple-choice knapsack problem. *Operations Research 27*, 3 (1979), 503–515. 5

[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, 9, 10

[Tre] CUDACOMPRESS compression library. https://github.com/m0bl0/cudaCompress. [accessed: 2019-02-01]. 9

[Tur] Johns Hopkins Turbulence Databases. http://turbulence.pha.jhu.edu/datasets.aspx. [accessed: 2018-10-31]. 1, 9

[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. 11

[Wes95] WESTERMANN R.: Compression domain rendering of time-resolved volume data. In *Proc.IEEE Vis* (1995), pp. 168–175. 2

[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. 2

[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. 1, 2

[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. 2

[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