Interactive Multiscale Tensor Reconstruction for Multiresolution Volume Visualization

Susanne K. Suter, *Student Member, IEEE*, José A. Iglesias Guitián, Fabio Marton, Marco Agus, Andreas Elsener, *Member, IEEE*, Christoph P.E. Zollikofer, M. Gopi, Enrico Gobbetti, and Renato Pajarola, *Member, IEEE*

Abstract—Large scale and structurally complex volume datasets from high-resolution 3D imaging devices or computational simulations pose a number of technical challenges for interactive visual analysis. In this paper, we present the first integration of a multiscale volume representation based on tensor approximation within a GPU-accelerated out-of-core multiresolution rendering framework. Specific contributions include (a) a hierarchical brick-tensor decomposition approach for pre-processing large volume data, (b) a GPU accelerated tensor reconstruction implementation exploiting CUDA capabilities, and (c) an effective tensor-specific quantization strategy for reducing data transfer bandwidth and out-of-core memory footprint. Our multiscale representation allows for the extraction, analysis and display of structural features at variable spatial scales, while adaptive level-of-detail rendering methods make it possible to interactively explore large datasets within a constrained memory footprint. The quality and performance of our prototype system is evaluated on large structurally complex datasets, including gigabyte-sized micro-tomographic volumes.

Index Terms—GPU/CUDA, multiscale, tensor reconstruction, interactive volume visualization, multiresolution rendering.

1 Introduction

The continuing advances in 3D imaging technologies, such as the improvements of phase contrast synchrotron and micro-computed X-ray tomography, as well as the ever higher resolution of numerical simulations brought up by high performance computing, are leading to the generation of large scale and structurally complex volume datasets. These large 3D data volumes not only represent an immense amount of information, but also exhibit an increasing level of detail of internal structure in space (and possibly time), resulting in a high degree of complexity at different scales. The visualization challenge is to create new methods that allow analysts to visually examine and understand these datasets of overwhelming size and complexity.

Direct volume rendering (DVR) is the technique of choice for interactive data visualization and exploration, since it supports the representation of the full dataset in a single image using a variety of semi-transparent mappings. In the last few years, improvements in programmability and performance of GPUs have made GPU solutions the main option for real-time rendering on desktop platforms [9]. However, the sheer size of nowadays large datasets requires the integration of adaptive data reduction methods based on levels of detail (LOD), together with out-of-core memory management techniques, since full datasets typically do not fit in the available GPU memory (nor main memory), and, even if they could, traversing all data voxels for computing volume integrals remains too costly for real-time rendering. The key point is to define a suitable multiresolution model, an approximation hierarchy over the volume that can be traversed at run-time to adaptively select a LOD, which satisfies a certain visual quality or rendering performance threshold for every displayed frame.

In recent years, much effort has been put into the development of mathematical representations that can approximate complex data. A widespread approach is to find a limited set of numerical bases together with a corresponding set of coefficients whose weighted linear combination is a close approximation of the original data. Decomposition of large datasets into bases has two main objectives: (1) save memory and bandwidth to accelerate rendering tasks, and (2) extract and represent relevant features, where relevance is defined by the scientific questions asked during the dataset visualization. The potential of these methods for simultaneous bandwidth reduction and feature extraction is largely unexplored. Accordingly, the goal is to search for feature-specific bases that reflect statistical (spatial) properties of the features to be extracted rather than to use a-priori bases.

Tensor approximation (TA) frameworks, as summarized in [15], are an extension of standard matrix data approximation tools, such as SVD, to higher-orders and provide such bases for a compact linear data representation (see Appendix A). There are two parts to tensor approximation: (1) *tensor decomposition*, usually an offline process, to compute the bases and coefficients, and (2) *tensor reconstruction* from the previously computed bases, which should be a fast online process in an interactive visualization application. Suter et al. [23] have recently presented preliminary results on small datasets, which indicate that TA promises to be a viable alternative to more traditional approaches like, e.g., wavelets, for creating compact feature-preserving volume representations. The *efficiency* of TA for interactive DVR, associated to real-time tensor reconstruction, is currently mostly unexplored, as is the applicability of such methods to large datasets.

Contributions: In this paper, we advance the state-of-the-art by introducing the first GPU accelerated integration of multiscale volume tensor reconstruction within a real-time single-pass multiresolution volume ray-casting framework. Our specific contributions are:

- we introduce and analyze an end-to-end system based on a bricked tensor decomposition, which is capable of pre-processing and rendering in real-time multi-gigabyte datasets using a limited memory footprint;
- we introduce and describe a CUDA TA reconstruction process with a computational complexity growing only linearly with the reduced tensor rank and with an implementation exploiting the parallelism and memory hierarchy of GPGPU platforms;
- we introduce and evaluate a tensor specific quantization scheme, which reduces the out-of-core memory footprint as well as disk to CPU to GPU data transfer bandwidth;
- we provide an experimental analysis of the impact of the above contributions on massive volume datasets (17GB).

- *S.K. Suter, A. Elsener, C.P.E. Zollikofer, and R. Pajarola are with University of Zurich, Switzerland, E-mail: susuter@ifi.uzh.ch, elsener@ifi.uzh.ch, zolli@aim.uzh.ch, and pajarola@acm.org.*
- *J.A. Iglesias Guitián, F. Marton, M. Agus and E. Gobbetti are with CRS4, Italy, E-mail: jalley@crs4.it, marton@crs4.it, magus@crs4.it, and gobbetti@crs4.it.*
- *M. Gopi is with University of California, Irvine, USA, E-mail: gopi@ics.uci.edu.*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

2 Related Work

LOD based rendering using hierarchical volume representations as well as multiresolution out-of-core models are standard techniques to achieve interactive visualization performance for large scale volume data [9]. Most modern techniques perform full ray traversal on the GPU. In this context, large volume data is handled by compressing it using adaptive texturing schemes to fit entire datasets into GPU memory [25], or by using flat [17] or hierarchical [12, 5, 14] multiresolution structures in conjunction with adaptive loaders to deal with datasets of potentially unlimited size. In this context, our contribution is the first integration of a GPU accelerated tensor reconstruction of multiscale volume data into a real-time and out-of-core LOD based volume renderer (i.e., MOVR [12, 14]).

Data reduction, in this context, is of great importance to save storage space at all stages of the processing and rendering pipelines, as well as to reduce time and cost of transmission between the layers of the memory hierarchy. Since lossless schemes typically provide limited gains [11], many of the efforts are concentrated around lossy approximation methods, combining various forms of data transformation and quantization. Fout and Ma [11] emphasize that compression and decompression processes have to be highly asymmetric: with todays GPU rendering capabilities, we can afford a slow (offline) compression, while, in contrast, decompression needs to be fast and spatially independent to enable real-time rendering. This fact rules out techniques for achieving higher compression ratios (e.g., complex variable-length entropy coders), but having detrimental effects on GPU decompression speeds.

Compact mathematical representations of volume datasets can be based on *predefined* or *data-specific* bases decompositions. Methods using predefined basis are often computationally cheaper, while methods using data-specific bases require more precomputing time (to compute the bases), but are potentially able to remove more redundancy from a dataset. Today, predefined methods like *Fourier transform* [4], *wavelet transform* (e.g., [21, 13, 11]), and *discrete cosine transform* [28], as well as data-specific bases like *vector quantization* [22, 11, 20] are well-known approaches for volume data representations. In particular, wavelet transform and vector quantization, often combined together, are standard tools for compression domain volume rendering. Wavelets are especially convenient for compressed LOD DVR since they define a multiresolution hierarchy of coefficients, where each coefficient improves the approximation – higher-level coefficients are more important and small coefficients may be thresholded. Similary, there exist hierarchical vector quantizers [22], where the focus lies on finding an optimal decoding scheme for a real-time and simultaneous GPU reconstruction.

Using predefined or data-specific bases should be seen as two alternatives with both assets and drawbacks. Wavelets correspond to spatial averaging and the wavelet coefficients are derived from the convolution of applying one-dimensional filters along the spatial axes. That makes it difficult to compactly represent unaligned three-dimensional features [23]. There has been much work on developing more powerful oriented wavelet bases for multi-dimensional spaces [7, 8]. However, such bases are still data-independent prescribed filters, and the gained compression efficiency over axis-aligned bases is limited [27].

We would like to have a fresh look at the problem by learning preferential bases directly from the data, thus removing a bias in the approximation process. This, so far, has been done in DVR using learned codebooks combined with vector quantization [22, 11, 20], which require, however, large dictionaries if low distortion is desired. While computational aspects and rendering performance of base decomposition methods have been studied extensively, these methods are only beginning to be used for feature extraction. Our choice of approach was, however, to find bases, which simultaneously reduce datasets and represent features.

The most common tools for data approximation with computed bases are the singular value decomposition (SVD) and the principal component analysis (PCA). Both approaches work on 2D matrix data and exploit the fact that the dataset can be represented with a few highly significant coefficients and corresponding reconstruction vectors (based on the matrix rank reduction concept). The extension of rank-reduced data approximation to higher-order is not unique and can be grouped into two main so called tensor approximation (TA) approaches: the Tucker model and the Candecomp/Parafac (CP) model, which were recently reviewed in [15].

Recently, tensor approximation has been demonstrated to be a valid alternative 3D spatial volume compression method [24, 27, 23]. Our work builds on the approaches of Wu et al. [27] and Suter et al. [23], where tensor approximation is used as an approach to reduce and visualize volume data with a mathematical representation. Specifically, we use a Tucker model for which a short summary is given in Appendix A. Properties of the Tucker model and common notations, which we follow in this work, are elaborated in [15]. It should be noted that the TA approach has the advantage that the ranks of the factor matrices (and in turn the size of the core tensor) can be conveniently reduced without destroying cache coherence and without data repacking.

Our idea was to extend and improve the offline TA method used in [23] to a real-time interactive system and verify it with large datasets. We extended the previous work [23] with a bricked TA implementation, we evaluated and defined a Tucker-specific quantization scheme and we provided a real-time tensor decomposition reconstruction on the GPU. As in other works, out-of-core [26] and hierarchical [27] data management techniques were applied to our approach.

A complete hierarchical multiscale TA system has been proposed in [27]. Their data hierarchy consists of tensor decomposed subvolumes with each level having a progressively decreasing rank-reduction level. All subvolumes or (volume) bricks – as they are called – on one hierarchy level, representing the residual to the parent level, are treated as a tensor ensemble. As a result, each hierarchy level consists of one single rank-reduced core tensor and multiple factor matrices. However, for interactive reconstruction and visualization many temporary results must be cached in the hierarchy at run-time. Instead, we build a multiresolution TA octree where each lower resolution brick is an autonomous tensor decomposition that can independently be reconstructed and rendered on demand, attaining interactive speed.

The reconstruction process from our chosen tensor decomposition is, in principle, straightforward, and can be optimized by a careful reordering of operations (Appendix A.2). Nevertheless, reconstruction time can be critical for real-time visualization and therefore needs to be given attention in our system. For the first time, we address GPGPU-based tensor reconstruction and therefore the reconstruction concepts should fit parallel computing concepts. E.g., while thresholding of tensor coefficients [27] may reduce the amount of data, the reconstruction process can be more tedious for parallel computing since complex decoding algorithms have to be used. Moreover, the reconstruction is affected by the format and representation of the coefficients.

For compact data representation, the encoding of numerical values is fundamental and hence an appropriate quantization must be devised. We refrain from variable-length coding at this point to avoid the corresponding costly decompression. Fixed linear quantization for the factor matrices (8-bit) and core tensor (8-20-bit) has been proposed in [27]. We investigate more tensor-specific linear as well as non-linear quantizations, suitable for fast reconstruction implementation on the GPU. In particular, the distribution of the core tensor coefficients can benefit from logarithmic quantization, and we analyze the error rate of different quantization strategies.

There exist only a few commonly available TA implementations: e.g, there is a comprehensive MatLab toolbox [2] , and for C++ there is a tensor framework for 3D datasets available with the VMMLib [1], which was extended for this project and used for the preprocessing part on the CPU. However, no hardware accelerated implementations have been proposed so far. In this paper, we present the first CUDA [18, 19] based GPU accelerated implementation of a reduced complexity Tucker volume tensor reconstruction algorithm.

3 Bricked Multiresolution TA

Large volumes cannot be processed/rendered as a monolithic block, since not only their size exceeds available working memory, but spatial adaptation is paramount for implementing fast renderers. Therefore, a data management system that divides the data into blocks is an important basis both to process and to visualize large datasets. Our method is based on the offline decomposition of the original volumetric dataset into small cubical bricks (subvolumes), i.e., third-order tensors, which are approximated, quantized and organized into an octree structure maintained out-of-core. The octree contains data bricks at different resolutions, where each resolution of the volume is represented as a collection of bricks in the subsequent octree hierarchy level.

Each brick has a fixed width B with an overlap of two voxels at each brick boundary for efficiently supporting runtime operations requiring access to neighboring voxels (trilinear interpolation and gradient computation). The width of the brick is flexible, but in this paper is set to $B = (28+2+2) = 32$, i.e., one brick is 32^3, which has proved small enough to guarantee LOD adaptivity, while coarse enough to permit an effective brick encoding by the analysis of the local structure.

Each octree brick $\mathscr{A} \in \mathbb{R}^3$ is tensor approximated using rank-reduced Tucker decomposition. A Tucker decomposition (see Appendix A) is defined as $\widetilde{\mathscr{A}} = \mathscr{B} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}$, where $\mathscr{B}$ is the so called core tensor and $\mathbf{U}^{(n)}$ are the factor matrices. A rank-reduced TA along every mode of the dataset is written with the notation: rank-(R_1, R_2, R_3) TA. As illustrated in Fig. 1, we compute for each brick of size B^3 a rank-(R,R,R) TA, with $R \in [1..B-1]$. Typically, we use a rank reduction, where $R = B/2$, i.e., $R = 16$ for $B = 32$, following the rank reduction scheme used in other tensor approximation works [27, 23]. The resulting rank-reduced decomposition is quantized to further reduce memory usage (see Sec. 4) and stored in a out-of-core brick database. With each brick, we store a 64-bit binary histogram, which is used for transfer-function-based culling.

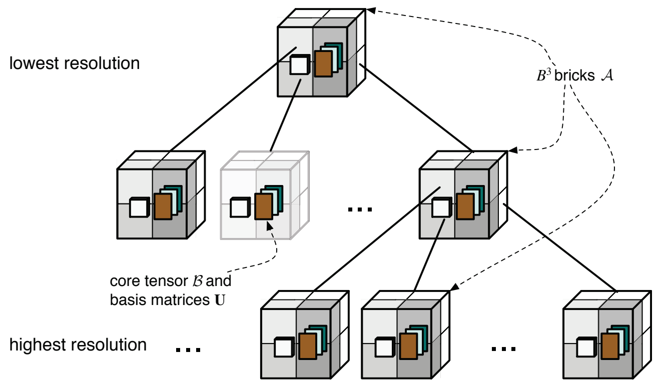

lowest resolution
B^3 bricks $\mathcal{A}$
core tensor $\mathcal{B}$ and
basis matrices $\mathbf{U}$
highest resolution
...
...
...

Fig. 1. Multiresolution octree tensor decomposition hierarchy with B^3 sized bricks.

The whole preprocessing is performed in a low-memory setting using a bottom-up process on a brick-by-brick basis, which is repeated until we reach the octree root. Leafs are constructed by sampling the original dataset, while non-leaf bricks are constructed from their previously constructed eight children, which are dequantized, reconstructed, and spatially averaged.

At run-time, an adaptive loader updates a view- and transfer function-dependent working set of bricks. The working set is incrementally maintained on the CPU and GPU memory by asynchronously fetching data from the out-of-core brick multiresolution TA structure. Following the MOVR approach [12, 14], the working set is maintained by an adaptive refinement method guided by the visibility information fed back from the renderer. The adaptive loader maintains on GPU a cache of recently used volume bricks, stored in a 3D texture. At each frame, the loader constructs a spatial index for the current working set in the form of an octree with neighbor pointers.

For rendering and visibility computation, the octree is traversed using a CUDA stack-less octree ray-caster, which employs preintegrated scalar transfer functions to associate optical properties to scalar values, and supports a variety of shading modes [14]. The ray-caster works on reconstructed bricks, and reconstruction steps occur only upon GPU cache misses. The quantized tensor decomposition is dequantized and reconstructed on demand by the adaptive loader during the visualization on the GPU (see Sec. 5).

In order to permit structural exploration of the datasets, the reconstruction can consider only the K most significant ranks of the tensor decomposition, where $K \in [1..R]$ is chosen by the user. The reconstruction rank K can be changed during the visualization process with a *rank slider*. Lower-rank reductions give a faster outline of the visualized dataset and can highlight structures at specific scales [23], see also Sec.6. Higher K values add more details onto the dataset.

4 Encoding of coefficients

As mentioned previously, the tensor and factor matrix coefficients take up unnecessary space if maintained as floating point values, see also storage cost analysis in Sec. 6.2. For compact representation of the tensor decomposition and to reduce the disk to host to device bandwidth during rendering, we apply a simple fixed bit length encoding based on tensor-specific quantization. In particular, the factor matrices and the core tensor of the Tucker model have a different distribution of coefficients and thus the quantization approach was selected accordingly, as described below. A fixed bit length approach has been selected in order to simplify parallel decoding on the GPU.

4.1 Factor Matrices and Core Tensor Coefficients

The coefficients of the basis *factor matrices* $\mathbf{U}^{(1\ldots3)}$ are normalized and distributed between $[-1,1]$, due to the orthonormality of factor matrices in the Tucker model. Therefore, a uniform linear 8- or 16-bit quantization as in Eq. 1 can effectively be applied. We use a single *min*/*max*-pair to indicate the quantization range for all three factor matrices to minimize the number of coefficients that need to be loaded by the CUDA kernels.

$$\tilde{x}_{\mathbf{U}} = (2^{Q_{\mathbf{U}}} - 1) \cdot \frac{x - x_{min}}{x_{max} - x_{min}} \tag{1}$$

As per definition of the Tucker model, the *core tensor* $\mathscr{B}$ captures the contribution of the linear bases combinations, i.e., the *energy* of the data, in its coefficients. The distribution of the signed coefficients is such that the first entry of the core tensor has an especially high absolute value close to the volume's norm, capturing most of the data energy, while many other entries concentrate around zero. The probability distribution of the other values between the two extrema is decreasing with their absolute magnitude in a logarithmic fashion. Hence we apply a logarithmic quantization scheme as in Eq. 2 for the core tensor coefficients, using a separate sign-bit.

$$|\tilde{x}_{\mathscr{B}}| = (2^{Q_{\mathscr{B}}} - 1) \cdot \frac{\log_2(1+|x|)}{\log_2(1+|x_{max}|)} \tag{2}$$

Special treatment is given to the one first high energy value mentioned before. It is known that this value, the *hot-corner coefficient*, is always at position $\mathscr{B}(0,0,0)$. Since it is one value and in order to give more space to the quantization range to the other coefficients, we optionally do not quantize this value and store it separately.

Various quantization levels for the other coefficients, $Q_{\mathbf{U}}$ and $Q_{\mathscr{B}}$, could be used and analyzed. In practice, we have chosen a byte-aligned quantization of $Q_{\mathbf{U},\mathscr{B}} = 8$- or 16-bit as a compromise between the most effective quantization and efficient bit-processing. The effects of quantization as well as other tensor-specific optimizations are reported in Sec. 6.2 where we analyze the quantization error.

4.2 Storage Requirements

The basic storage needed for a volume dataset $\mathscr{A}$ of size of $I_1 \times I_2 \times I_3$, is $I_1 \cdot I_2 \cdot I_3 \cdot Q$, where Q is the number of bits (bytes) per scalar value. A rank-(R_1,R_2,R_3) tensor approximation, however, only requires $R_1 \cdot R_2 \cdot R_3 \cdot Q_{\mathscr{B}} + (I_1 \cdot R_1 + I_2 \cdot R_2 + I_3 \cdot R_3) \cdot Q_{\mathbf{U}}$, in addition to three floating point numbers for the quantization ranges of the factor matrices (*min*/*max* values) and core tensor (*max* quantization value), and one floating point value for the hot-corner value. This first coefficient of the core tensor is (optionally) encoded separately from the remaining ones, leading to a reduced quantization range for Eq. 2.

The ratio of the achieved data reduction depends on the input size and rank reduction of the TA. In Sec. 6.2, we analyze the amount of storage needed for rank-reduced and quantized TA of complete data volumes as well as bricked multiresolution volumes.

5 GPU Tensor Reconstruction

For volume ray-casting one can consider either a per-voxel (e.g., for random access during traversal) or per-brick reconstruction approach. Using a per-brick solution permits us to optimize reconstruction by refactoring computations in order to reduce computational costs and to take advantage of the complex memory hierarchy of nowadays GPGPU platform. As outlined in the previous section, we perform a brick-wise reconstruction of the tensor from its decomposition according to the multiresolution octree hierarchy. If not already cached by the rendering system, a requested tensor brick is loaded and transferred to the GPU where the tensor reconstruction is performed. In the following we analyze the basic tensor reconstruction strategy and its implementation on the GPU using the CUDA framework.

5.1 CUDA Terminology

A CUDA kernel is a SIMD parallel program that is executed by an array of *threads* (work-items),[1] all running the same code. The threads are organized in a grid of thread blocks, and each kernel is called with a given *grid* size (NDRange) and a given *blocks* size (work groups). Each thread owns some *registers* (private memory) and each thread block has access to a limited amount of *shared memory* (local memory), which constitute the fastest data access paths. Additionally all threads can concurrently access *global memory*, *constant memory* and *texture memory* (global memory), where the memory latency increases from constant and texture memory (read only and cached), to local and global memory (read/write).

5.2 Reconstruction Strategy

First, we describe a simple but not optimized implementation to set out a starting approach for the reconstruction problem. Later, we introduce an optimized reconstruction based on the Tucker formulation implemented as so called *tensor times matrix* (TTM) multiplications. In both cases, the reconstruction on the GPU is parallelized such that the voxels of each brick are reconstructed in parallel, i.e., one thread of the GPU kernel computes one voxel. According to Eq. A.5 the simple reconstruction solution can be implemented with a single CUDA kernel that sums over all core tensor coefficients, each multiplied with the corresponding entries from the factor matrices. Despite the simplicity of this implementation drawbacks arise as a consequence. First, Eq. A.5 includes a triple-for-loop, which makes the computational cost per reconstructed voxel and thread of cubic order R^3, where R is the number of reduced ranks. Second, to reconstruct a single voxel the complete core tensor $\mathscr{B}$ needs to be available in each thread, which is not memory efficient.

The computational complexity can significantly be reduced by sequentially applying the core tensor and factor matrix multiplications using n-mode products as in Eq. A.6 and storing intermediate reconstruction results. This reconstruction corresponds to the initial formulation in Eq. A.3 and has been implemented as so called *tensor times matrix* (TTM) multiplications. With this three step TTM volume reconstruction, the computational complexity per reconstructed voxel grows only linearly with R. Hence, we have to call three different GPU TTM kernels successively and store the intermediate results in between kernel calls. The CUDA implementation of this approach is described in more detail below.

5.3 CUDA TTM Reconstruction

The tensor reconstruction in CUDA is implemented using three successively applied TTM kernels. Each kernel corresponds to the application of one n-mode product (as in Eq. A.6), hence TTM1, TTM2, and TTM3. After applying TTM1 and then TTM2, we need each time to temporarily store a third-order tensor of size $B \times R^2$ and $B^2 \times R$, respectively. Eventually after applying TTM3 the final B^3 sized volume brick is reconstructed.

[1]Analogous OpenCL terms are mentioned in parenthesis.

The implementation of TTMn: $\mathscr{Y} = \mathscr{X} \times_n \mathbf{U}^{(n)}$ is based on a matrix-matrix multiplication as illustrated in Fig. 14. More precisely, the factor matrix $\mathbf{U}^{(\mathbf{n})}$ is multiplied with each slice (matrix) of the third-order tensor $\mathscr{X}$, sliced according to the mode n (e.g., lateral, frontal, or horizontal slices). Hence the full reconstruction $\widetilde{\mathscr{A}} = \mathscr{B} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}$ corresponds to successive matrix-matrix multiplications, which can be optimized by following CUDA implementation best practices [18].

The pseudo code implementation of the three CUDA kernels is given in the Algs. 1–3. We use a thread block per decoded brick, where each thread is responsible for computing one element of the tensor-matrix multiplication. The grid-size for parallel execution is determined by the number of bricks that need to be decoded in the current frame (e.g., 8 bricks for the minimal octree refinement step).

For TTM1, we compute one slice of the intermediate tensor $\mathscr{B}'$ on one thread block. For one TTM1-block, we load the factor matrix $\mathbf{U}^{(\mathbf{1})}$ and one slice of $\mathscr{B}$ (slice is given by blockId). For reasons of memory optimizations, we compute for TTM2 and TTM3 only half slice of $\mathscr{B}''$ and half slice of $\widetilde{\mathscr{A}}$, respectively, on one thread block. For one TTM2/TTM3-block, we load half of the factor matrix and one slice of the intermediate data structures $\mathscr{B}'$ and $\mathscr{B}''$. The memory usage and performance optimizations of the CUDA TTM reconstruction are explained next.

Algorithm 1 CUDA kernel for TTM1

```
1: load $\mathbf{U}^{(\mathbf{1})}$ and tensor core $\mathscr{B}$ slices to GPU
2: CUDA kernel:
3: extract min/max values for dequantization
4: linearly dequantize one element of $\mathbf{U}^{(\mathbf{1})}$
5: log dequantize one element of $\mathscr{B}$
6: each thread writes one element of $\mathbf{U}^{(\mathbf{1})}$ to shared memory
7: each thread writes one element of the $\mathscr{B}$_slice to shared memory
8: __synchthreads()
9: for each r1 in R1 do
10:     voxel += $\mathbf{U}^{(\mathbf{1})}(i1, r1) \cdot \mathscr{B}(r1, r2, r3)$
11: end for
12: store voxel to the intermediate $\mathscr{B}'(i1, r2, r3)$
```

Algorithm 2 CUDA kernel for TTM2

```
1: load $\mathbf{U}^{(\mathbf{2})}$ and half tensor $\mathscr{B}'$ slices to GPU
2: CUDA kernel:
3: extract min/max values for dequantization
4: linearly dequantize one element of $\mathbf{U}^{(\mathbf{2})}$
5: each thread writes one element of $\mathbf{U}^{(\mathbf{2})}$ to shared memory
6: each thread writes one element of the $\mathscr{B}'$_slice to shared memory
7: __synchthreads()
8: for each r2 in R2 do
9:     voxel += $\mathbf{U}^{(\mathbf{2})}(i2, r2) \cdot \mathscr{B}'(i1, r2, r3)$
10: end for
11: store voxel to the intermediate $\mathscr{B}''(i1, i2, r3)$
```

Algorithm 3 CUDA kernel for TTM3

```
1: load $\mathbf{U}^{(\mathbf{3})}$ and half tensor $\mathscr{B}''$ slices to GPU
2: CUDA kernel:
3: extract min/max values for dequantization
4: linearly dequantize one element of $\mathbf{U}^{(\mathbf{3})}$
5: each thread writes one element of $\mathbf{U}^{(\mathbf{3})}$ to shared memory
6: each thread writes one element of the $\mathscr{B}''$_slice to shared memory
7: __synchthreads()
8: for each r3 in R3 do
9:     voxel += $\mathbf{U}^{(\mathbf{3})}(i3, r3) \cdot \mathscr{B}''(i1, i2, r3)$
10: end for
11: add contribution of hot-corner core value to voxel
12: store voxel in decoding buffer for $\widetilde{\mathscr{A}}(i1, i2, i3)$
```

5.4 Performance Optimizations

In order to optimize the parallel execution on the GPU, we should exploit data parallelism, optimize memory usage, take into account the bandwidths to the various parts of the memory hierarchy, and optimize instruction usage, thus increasing throughput.

In our approach, host-to-device data transfers are reduced by using page-locked buffers. That is, before launching the kernel TTM1, we transfer the factor matrices and the core tensor of one brick all at once and in a quantized form to the global GPU memory. Our GPU TTM reconstruction code (TTM kernels) uses intermediate data structures ($\mathscr{B}'$ and $\mathscr{B}''$), which allows us to make use of the on-chip memory.

The temporary data structures of the third-order tensors are stored in the global memory, and slices of these third-order tensors are loaded to the shared memory when requested. Threads within the same thread block cooperate to load into shared memory the necessary elements of $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$,$\mathbf{U}^{(3)}$, $\mathscr{B}$, $\mathscr{B}'$ and $\mathscr{B}''$, by loading one single element per tensor per thread. A *syncthreads()* barrier at the end of the loading phase ensures that all the elements are up-to-date before performing calculations. In order to avoid bank conflicts in shared memory accesses, we dequantize the factor matrices and core tensor to 32-bit floating point words before we upload the data to shared memory. Thus, all accesses are aligned on 32-bit words.

For TTM2 and TTM3, we split the matrix-matrix multiplication of one slice into two blocks. In that way, we optimize the shared memory usage and load only half of a factor matrix together with the full core tensor (we thus compute only upper or lower half of a matrix with the other matrix). With this scenario, we need for TTM1 $(B \cdot R \cdot 4 + R \cdot R \cdot 4)$ bytes and for TTM2/TTM3 $(B/2 \cdot R \cdot 4 + B \cdot R \cdot 4)$ bytes of shared memory per block, which works with 32-cubed bricks for CUDA 1.x and 2.x. We have a maximum of 16 KB (CUDA 1.x) or 48 KB (CUDA 2.x) of shared memory available per multiprocessor, whereas one multiprocessor can have at maximum 8 blocks. Depending on how much shared memory is used, fewer or more blocks are loaded per multiprocessor. To summarize, in our implementation, increasing the use of shared memory has higher priority than maximizing the number of blocks run per processor (for CUDA 1.x).

In addition, TTM3 separately handles the contribution of the *hot-corner coefficient*, adding its contribution to the overall reconstruction directly from the unquantized value of $\mathscr{B}(0,0,0)$. In order to simplify decoding and avoid special case handling in the inner loop, we encode a zero quantized coefficient at the corresponding core tensor position.

The final reconstruction $\widetilde{\mathscr{A}}$, is stored into an intermediate decoding buffer in device linear memory. We perform for each decoded brick a device-to-device copy in order to place the decoded brick in the correct place in the GPU cache, which is maintained as a texture bound to a CUDA array in order to maximize texture memory cache coherence.

6 Experimental Results

We have implemented a library supporting the multiscale volume tensor reconstruction and its integration with a GPU-accelerated out-of-core multiresolution volume rendering framework using C++ and CUDA 3.2. All performance tests have been carried out on an Intel Core 2 E8500 3.2GHz Linux PC with 4GB RAM, and GeForce GTX 480 graphics with 1.5GB of memory. The multiresolution volume octree is stored in an out-of-core structure, based on the Berkeley DB, with each 32^3 brick being stored as a quantized rank-$(16,16,16)$ tensor decomposition.

We discuss the performance results obtained based on the large scale micro-CT veiled chameleon and great ape molar (tooth) datasets. The chameleon ($1024^2 \times 1080$, 16-bit) is a micro-CT scan of the upper body, where each slice is $0.105mm$ thick with an inter-slice spacing of $0.105mm$ and a field of reconstruction of $94.5mm$. The tooth (2048^3, 16-bit) is a $7mm^3$ block cut out of dental enamel, scanned by phase-contrast synchrotron tomography at high resolution to reveal growth patterns of the dental enamel. Additionally, we use three smaller datasets ($256^2 \times 128$) for the quantization error analysis.

Preprocessing consisted in the construction and storage of the multiresolution volume octree, including the computation of the tensor decomposition for all bricks and the quantization of the coefficients. The preprocessing time for the 2GB chameleon dataset was $25min15sec$ and produced a 231M file. In the case of the 17GB tooth dataset the preprocessing time was $8h45min$ and the resulting file was of 5.5GB.

6.1 Interactive Performance

We evaluated the rendering and tensor reconstruction performance on the two large volumes (Figs. 2 and 3). The qualitative performance and interactivity of our adaptive GPU ray-caster is demonstrated in an accompanying video, recorded using a window size of 1024×768 pixels using a 1 voxel/pixel LOD rendering accuracy threshold. Our interactive inspection sequences include overall views and extreme close-ups, which stress our adaptive loader by incrementally requesting and reconstructing a large number of bricks.

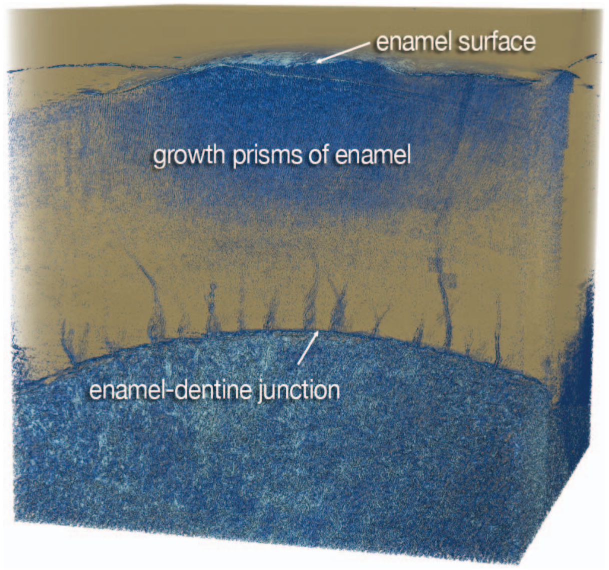


Fig. 2. Great ape molar (tooth), $7mm^3$ sampled at 2048^3, scanned with phase-contrast synchrotron tomography.

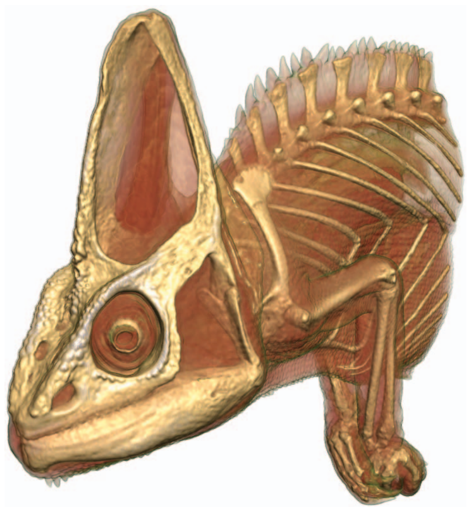
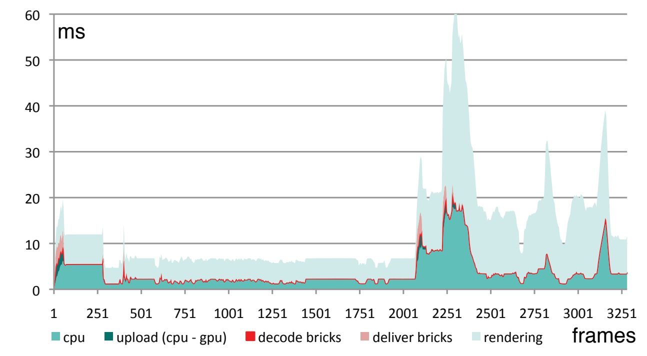
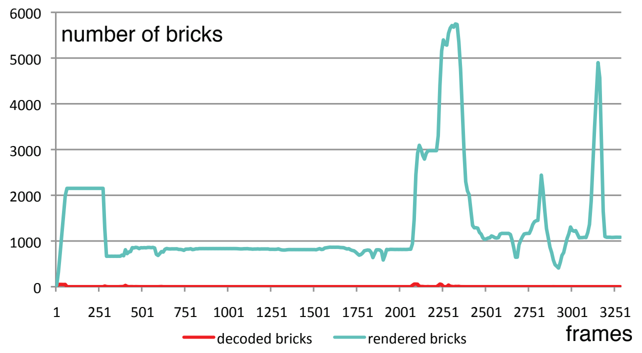


Fig. 3. Full reconstruction of the chameleon.

Figs. 4 and 5 demonstrate the achieved performance. As we can see, in any case an interactive rendering performance can be maintained, with frame-rates higher than 12Hz even for the most demanding situations, and on average between 50Hz and 100Hz. In particular, the timing reveals that our tensor reconstruction constitutes only a negligible overhead with respect to the overall rendering cost. Rendering time is in fact dominated by the ray-casting and data transfer times. The most costly part of the (fast) tensor reconstruction process is the final copy of the decoded bricks to texture cache.

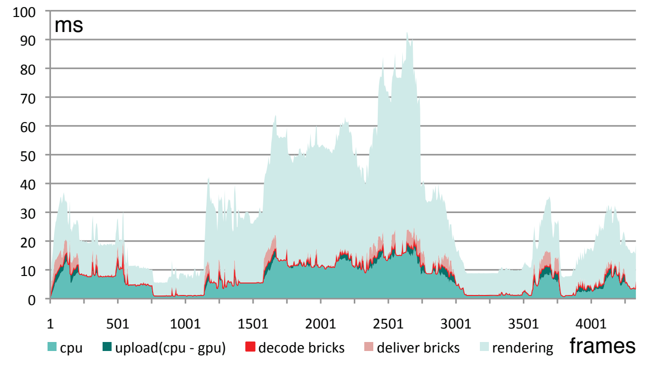


(a) Performance in ms per frame

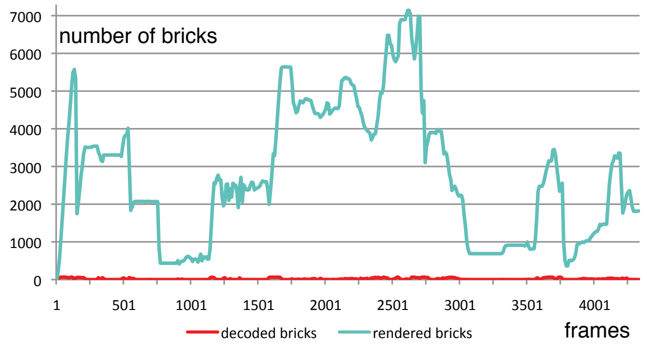


(b) Number of bricks per frame

Fig. 4. Performance measured on the chameleon.



(a) Performance in ms per frame



(b) Number of bricks per frame

Fig. 5. Performance measured on the great ape molar (tooth).

The number of rendered bricks per frame varies depending on the zoom factor, and is always maintained below 7000 by our adaptive renderer. Brick dequantization and reconstruction occurs only upon cache misses, which attributes to the low tensor reconstruction cost. But even under the most stressful situations where the number of rendered bricks changes rapidly, the dynamic update process is largely dominated by the brick data uploading time from CPU to GPU and not by the tensor reconstruction.

Given the high tensor reconstruction performance, we see as an interesting avenue for future work the possibility to further reduce device memory occupation by removing the uncompressed brick cache and decoding bricks on-demand at each frame. This would allow to display even larger amounts of volume data in each rendered image.

6.2 Data Reduction

The quantization of the tensor coefficients helps to keep the critical CPU-to-GPU data transfer and disk storage low. In this section, we analyze the error due to quantization and how the storage size is thus affected. We considered quantization approaches that use the same bit-length (from 8-bit to 16-bit) for all values within a coefficient type, the factor matrices $\mathbf{U^{(n)}}$ and the core tensor $\mathscr{B}$.

6.2.1 Storage Cost

The storage cost for different quantization approaches is indicated in Fig. 6, where U and B indicate factor matrices or core tensor settings, respectively, and *k*lin/log indicates linear or logarithmic quantization to $Q_{\mathbf{U},\mathscr{B}} = k$ bits according to Eqs. 1 and 2. The left-most value A:16 represents the size of a 2048^3 16-bit input volume dataset $\mathscr{A}$, and U:32 B:32 a 32-bit floating point representation of the reference rank-$(1024, 1024, 1024)$ reduced tensor approximation of $\widetilde{\mathscr{A}}$. The data reduction follows the storage requirements outlined in Sec. 4.2.

We can see in Fig. 6 that the proposed quantization (U:8 B:8 to U:16 B:16) has a additional storage reduction effect, compared to the floating point tensor (U:32 B:32) and original volume (A:16) data representation. Furthermore, for the quantized 32^3-bricked multiresolution octree hierarchy the storage consumption is minimally different from the non-bricked quantized format. Only the non-quantized bricked floating-point representation has an adverse space cost behavior due to its many coefficients that have to be stored. The approximation quality of the different quantization levels is analyzed below. From the storage cost results we can conclude that it is preferable to spend 16-bits on the factor matrix entries rather than on the core tensor, as the factor matrices $\mathbf{U}^{(n)}$, being quadratic, affect the total storage marginally compared to the core tensor $\mathscr{B}$, being cubic (see Sec. 4.2).

6.2.2 Quantization Error

To evaluate the approximation quality of a rank-reduced and quantized tensor decomposition we use the *signal-to-noise ratio* (SNR) to express the error in relation to the data's signal strength. We define the signal strength of a volume $\mathscr{A}$ as the *averaged Frobenius norm* $\|\mathscr{A}\|_{\bar{F}} = \sqrt{\frac{1}{N}\sum a^2_{i_1,i_2,i_3}}$, and the approximation and quantization noise of the reconstructed volume $\widetilde{\mathscr{A}}$ as the *root-mean-squared error* (RMSE) $\varepsilon_{\widetilde{\mathscr{A}}} = \sqrt{\frac{1}{N}\sum(a_{i_1,i_2,i_3} - \tilde{a}_{i_1,i_2,i_3})^2}$. Hence the SNR is defined as $\sigma_{\widetilde{\mathscr{A}}} = 20 \cdot \log 10 \frac{\|\mathscr{A}\|_{\bar{F}}}{\varepsilon_{\widetilde{\mathscr{A}}}}$.

As base reference to evaluate quantization effects, we compare to the error which was introduced by a reduced rank-(R_1, R_2, R_3) tensor approximation $\widetilde{\mathscr{A}}$ of the original volume $\mathscr{A} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, where $R_k = \frac{I_k}{2}$ (following the rank scheme used in [27, 23]). Due to limits in processing time, the costly approximation error analysis was not performed on the full size 2048^3 tooth volume, but on a representative $256^2 \times 128$ subvolume, and put in comparison to the approximation quality achieved for the well known bonsai tree and engine datasets of the same resolution. The triple-bars in Fig. 7 are organized in the indicated dataset order and bright-dark-medium-luminance color coded. The reference floating-point tensor decomposition (U:32 B:32) is shown to isolate and evaluate the quantization effect.

Tucker TA-specific Quantization: We analyzed the quantization approaches outlined in Sec. 4.1, applying linear and logarithmic quan-

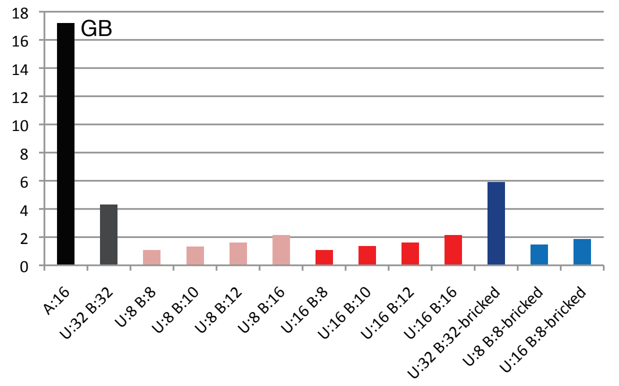


Fig. 6. Storage needed (in GB) for the various quantization approaches. U stands for the factor matrices, B for the core tensor. The number after B and U gives the number of bits used for the respective coefficient type.

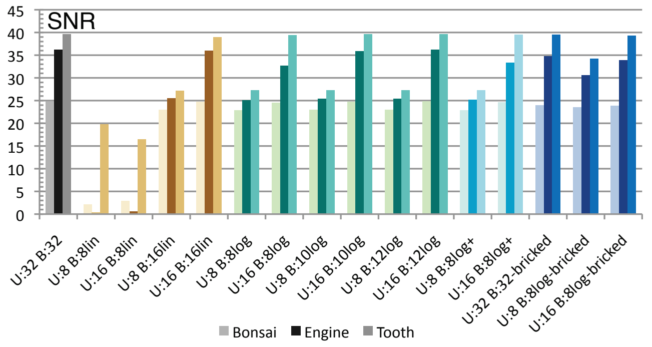


Fig. 7. Quantization error as SNR for various quantization approaches. The triple-bars are organized in the indicated dataset order and bright-dark-medium-luminance color coded. U stands for the factor matrices, B for the core tensor. The number after B and U gives the number of bits used for the respective coefficient type.

tization to both the factor matrices and core tensor as well. The effects on approximation error were analyzed for entire as well as bricked volume Tucker decompositions. Fig. 7 shows the analysis of the approximation quality in terms of the SNR $\sigma_{\widetilde{\mathscr{A}}}$ for different linear and logarithmic quantizations.

Except for the 8-bit linear core quantization (B:8lin), it is clear from Fig. 7 that for the factor matrices a significant improvement in SNR, and hence lower approximation error, can be achieved when using 16-bit (U:16) instead of 8-bit (U:8) quantization. Though the bonsai tree dataset does not benefit as strongly from this as the other volumes.

With respect to the core tensor quantization, it can be seen that the logarithmic is superior to the linear quantization, reaching comparable SNR values using much fewer bits, i.e. B:8log achieving almost the same quality as B:16lin for the same factor matrices quantization. It can be seen that increasing the quantization resolution from 8 to 12-bit only minimally improves the SNR, with the latter (B:12log) basically matching the more costly linear quantization.

We evaluated the separate floating point representation of the hot-corner core tensor coefficient (B:8log+ in Fig. 7), which otherwise potentially wastes quantization resolution better spent on the remaining core tensor coefficients. The SNR can so be increased slightly at the expense of only 4 extra bytes.

Taking the results from the storage cost study into account, the optimally compact quantization can be achieved using 16-bit linear factor matrix and 8-bit logarithmic core tensor quantization with separate hot-corner (U:16 B:8log+).

Bricked TA Quantization: In a bricked multiresolution octree setting the quantization quality differs only so slightly as shown in Fig. 7 (U:.. B:..-bricked), sometimes even being better. This could be explained by the fact that the bricked representation uses more coefficients in total over all bricks for the same volume dataset, consuming a little bit more space (Fig. 6). The preferable optimal quantization setting is thus the same as above also for the bricked TA.

Additionally, for the structural tooth dataset we performed a quantization error analysis on a larger 1024^3-cubed volume. The reference SNR of 37.16 for U:32 B:32 compares well with the SNR of 37.09 for U:16 B:8log-bricked, which matches well with the SNR study for the smaller tooth subvolume in Fig. 7. The SNR over individual bricks varies from 35.11 to 40.94 with an average of 37.16. Note that the U:16 B:8log-bricked representation differs from the original 16-bit input volume only by a very low (normalized) RMSE of 0.007.

6.3 Visual Results

In order to give insight into the capability of multiscale tensor approximation for DVR, we show visual results from the veiled chameleon and the great ape molar.

Approximative Visualization: The veiled chameleon dataset as shown in Fig. 8 is visualized with the out-of-core multiresolution volume renderer based on tensor reconstructed bricks. It can be seen that even with a lower rank tensor approximation, i.e., by using less storage and bandwidth, the essential parts as well as details of a certain feature size can be visualized. Small scale features are effectively removed from the visible bone structures, in a different way than by reducing the rendering LOD which typically results in a more blurred volume close up. Fig. 9 shows the effects of rank reduction on gradient quality. As we can see, block boundaries become apparent only at low ranks. Such artifacts are inherent to all brick-based lossy compression methods, and can be alleviated, at the cost of higher rendering time, by interblock interpolation through sampling neighboring bricks [16, 3] or by using deferred filtering approaches [10, 11]. This is orthogonal to the presented research.

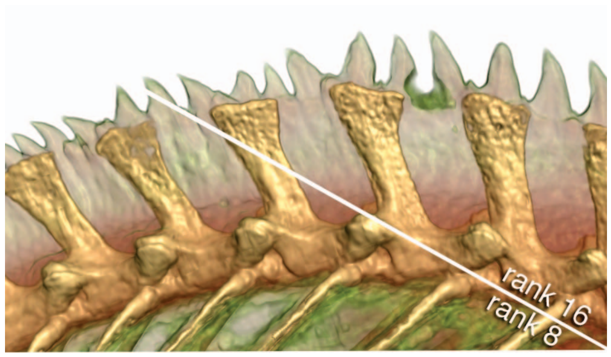


Fig. 8. Comparison of a rank-$(16,16,16)$ and a rank-$(8,8,8)$ TA.

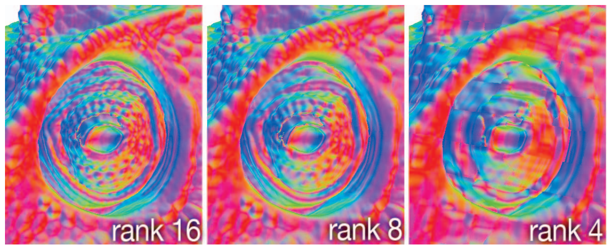


Fig. 9. Comparison of various rank-(R,R,R) TAs using the gradient vector of the skin isosurface mapped to RGB color.

Structural Features: As an application, we look at dental internal structures of a great ape molar. The relevant growth structures are found in the tooth enamel, which has a microstructure that is roughly comparable to densely packed fibers (so called prims). During dental enamel formation, each enamel prism elongates in centrifugal direction through the daily apposition of a small segment of enamel. In other words, the prisms grow along all three spatial dimensions and are in particular not axis-aligned, but of curved shape. Examples of such enamel growth patterns can be found in Suter et al. [23] and in the accompanying video. Those prisms represent important growth structures, but are difficult to visualize since the scanned high-resolution dataset includes spatial information of growth patterns that are of multiple scales (daily appositions form prisms).

In our experiments, we observed the effect of different rank-reduced and tensor-approximated dental growth structures in the great ape molar. We noticed that by using lower-rank TA, dental structures like growth prisms become highlighted as illustrated in an example close-up in Fig. 10. In Fig. 11 a horizontal cut orthogonal to the growth prisms (yellow dots) is shown. The image of the base reference at a rank-$(16,16,16)$ TA shows the prisms irregular spatial distribution, which makes the identification of individual prisms more difficult. The lower-scale rank-$(8,8,8)$ reconstruction clears out the fuzziness and reveals the layered periodic and parallel arrangement of the prisms. From these experiments, we conclude that the effect of rank-reduced TA supports counting or analyzing layer formations.

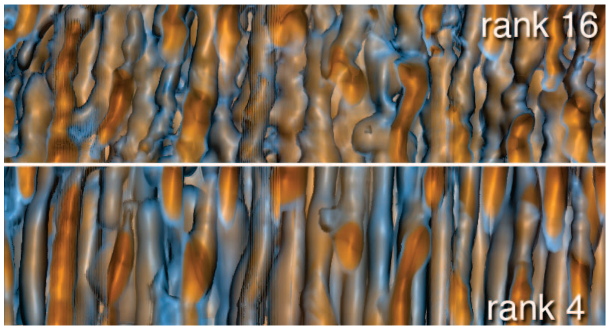


Fig. 10. Dental growth structures (prisms), highlighted with a reduced rank-$(4,4,4)$ reconstruction. Taken from a frontal projection to an area below the enamel surface (see Fig. 2).

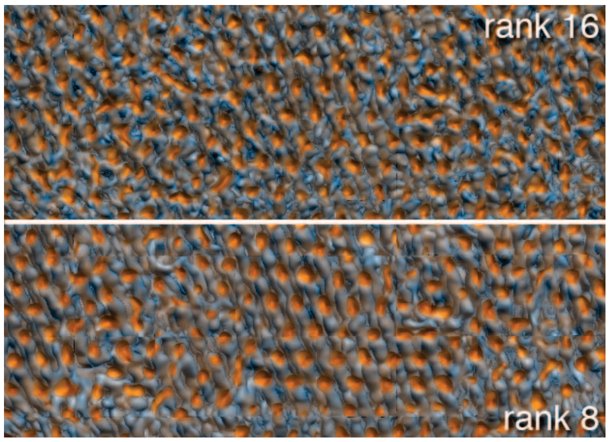


Fig. 11. Dental growth structures (prisms), highlighted with a reduced rank-$(8,8,8)$ reconstruction. Taken from a horizontal cut through an area below the enamel surface (see Fig. 2).

Even if more experimentation is required on a large variety of real-world datasets, our initial results seem to indicate that TA is able to preserve important features using coarse ranks. We see tensor-reconstructed volumes as an alternative to potentially help researchers to visualize and explore particular features at different scales by playing with tensor approximations of different ranks.

7 Conclusion

We have presented the first integration of a multiscale volume representation based on tensor approximation within a GPU-accelerated out-of-core multiresolution rendering framework. Our multiscale representation allows for the extraction, analysis and display of structural features at variable spatial scales, while adaptive level-of-detail rendering methods make it possible to interactively explore large datasets within a constrained memory footprint, allowing analysts to visually examine and understand datasets of overwhelming size and complexity. We have shown that tensor approximation offers good compression, and, by reducing the reconstruction rank, permits the highlighting of structural features. Thus, TA is a powerful approach to represent microstructural volume datasets at high data reduction ratios, and simultaneously highlighting relevant features at different spatial scales but high display resolution. Our system allows this exploration to occur for massive volumes and in real time.

Our future work will concentrate on further improving the performance and capabilities of our system by removing the need for an uncompressed brick cache, further reducing GPU memory needs. Moreover, we plan to improve our representation by using a per-brick adaption of the approximation rank, non-uniform quantization of coefficients, as well as thresholding of insignificant core tensor coefficients (sparse tensors) to further reduce memory needs. However, we would need to evaluate such a scenario with respect to the decoding and reconstruction times on the GPU.

Appendix A Tensor Approximation (TA)

In *tensor approximation* (TA) approaches, an multi-dimensional input dataset in array form, i.e., a tensor, is factorized into a sum of rank-one tensors or into a product of a core tensor and matrices, i.e., one for each dimension. This factorization process is generally known as tensor decomposition, while the reverse process of the decomposition is the tensor reconstruction. In the following sections, we give an overview of these two processes. In order to obtain a data reduction, which is an approximation of the input data, an additional process has to be introduced: the tensor rank-reduction. The concepts presented in the following subsections hold for general higher-order tensors. However, we restrict ourselves to third-order tensor as this is more intuitive and this represents the datasets used in *Direct Volume Rendering* (DVR).

A.1 Tensor Decomposition

Let $\mathscr{A}$ be a third-order tensor in $\mathbb{R}^{I_1\times I_2\times I_3}$ with elements $a_{i_1i_2i_3}$. The Tucker model is a common approach for tensor decompositions. There the third-order tensor is approximated by a product of a core tensor $\mathscr{B}$ and three factor matrices $\mathbf{U}^{(\mathbf{n})}$

$$\widetilde{\mathscr{A}} = \mathscr{B}\times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}, \qquad (A.3)$$

where the products $\times_n$ denote the n-mode product between the tensor and the matrices (see [15] and Fig.12). The Tucker decomposition of a tensor, which is a higher-order form of a matrix SVD or a PCA (extension of matrix rank concept), can be obtained from an *higher-order SVD* (HOSVD) algorithm (computed by a matrix SVD along every data mode). Other successful factorization methods are TUCKALS3 (an ALS approach for Tucker decompositions in three dimensions) and its generalized version, the *higher-order orthogonal iteration* (HOOI) (details see [15]).

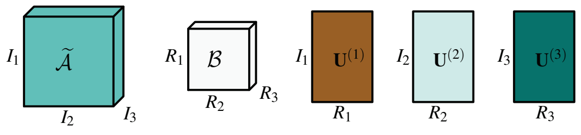


Fig. 12. Tensor decomposition

After a HOSVD the core tensor $\mathscr{B}$ has the same size as the original input dataset $\mathscr{A}$ and all the factor matrices are quadratic. However, we are more interested in light-weight, approximative Tucker decompositions, where $\mathscr{B}$ is element of $\mathbb{R}^{R_1\times R_2\times R_3}$ with $R_1 < I_1$, $R_2 < I_2$ and $R_3 < I_3$. Using the HOOI algorithm one can directly obtain a reduced-rank Tucker decomposition, whereas the rank reduction. Alternatively one can truncate the result obtained from HOSVD which is according to Bader and Kolda [15] not optimal in terms of difference between approximated and original data.

A.2 Tensor Reconstruction

The *tensor reconstruction* of a reduced-rank Tucker decomposition can be achieved in different ways. One alternative, is a progressive reconstruction: Each entry in the core tensor $\mathscr{B}$ is considered as weight for the outer product between the corresponding column vectors in the factor matrices

$$\widetilde{\mathscr{A}} = \sum_{r_1}\sum_{r_2}\sum_{r_3} b_{r_1 r_2 r_3} \cdot \boldsymbol{u}_{r_1}^{(1)} \cdot \boldsymbol{u}_{r_2}^{(2)} \cdot \boldsymbol{u}_{r_3}^{(3)}. \tag{A.4}$$

The sum of all theses weighted "subtensors" forms the approximation $\widetilde{\mathscr{A}}$ of the original data $\mathscr{A}$ (see Fig. 13).

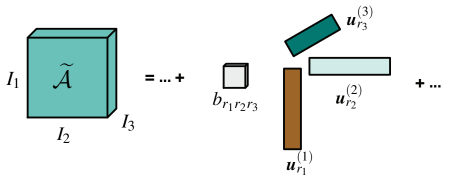

I_1
$\widetilde{\mathcal{A}}$
I_2
I_3
= ... +
$b_{r_1 r_2 r_3}$
$\boldsymbol{u}_{r_3}^{(3)}$
$\boldsymbol{u}_{r_2}^{(2)}$
$\boldsymbol{u}_{r_1}^{(1)}$
+ ...

Fig. 13. Tensor reconstruction from Eq. A.4 visualized.

Another approach, is to reconstruct each element of the approximated dataset individually, which we call *voxel-wise reconstruction* approach. Each element $\widetilde{a}_{i1i2i3}$ is reconstructed as shown in Eq. A.5, i.e., sum up all core coefficients multiplied with the corresponding coefficients in the factor matrices (weighted product).

$$\widetilde{a}_{i_1 i_2 i_3} = \sum_{r_1}\sum_{r_2}\sum_{r_3} b_{r_1 r_2 r_3} \cdot u_{i_1 r_1}^{(1)} \cdot u_{i_2 r_2}^{(2)} \cdot u_{i_3 r_3}^{(3)} \tag{A.5}$$

A third reconstruction approach is to build the n-mode products along every mode [15] (notation: $\mathscr{B} \times_n \mathbf{U}^{(n)}$), which leads to a tensor times matrix (TTM) multiplication for each mode, i.e., dimension. This is analogous to the Tucker model given by Eq. A.3. The n-mode product between a tensor and a matrix is equivalent to a matrix product as it can be seen from Eq. A.6. In Fig. 14 we visualize the TTM approach using n-mode products.

$$\mathscr{Y} = \mathscr{X} \times_n \mathbf{U} \Leftrightarrow \mathbf{Y}_{(\mathbf{n})} = \mathbf{U}\mathbf{X}_{(\mathbf{n})}, \tag{A.6}$$

where $\mathbf{X}_{(\mathbf{n})}$ represents the mode-n unfolded tensor, i.e., a matrix.

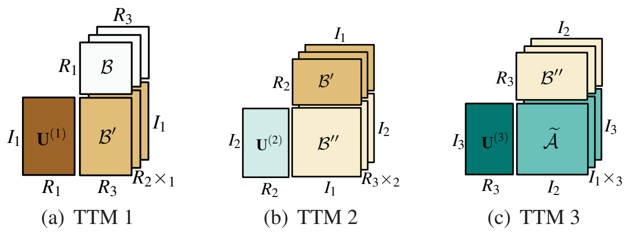


Fig. 14. TTM: tensor times matrix along the 3 modes (n-mode products). Backward cyclic reconstruction after Lathauwer et al. [6].

Given the fixed cost of generating an $I_1 \times I_2 \times I_3$ grid, the computational overhead factor varies from cubic rank complexity $R_1 \cdot R_2 \cdot R_3$ in the case of the progressive reconstruction (Eq. A.4) to a linear rank complexity R_1 for the TTM or the n-mode product reconstruction (Eq. A.5). (For $R = 16$, the improvement to $R^3 = 4'096$ is 256-fold.)

Acknowledgments

This work was supported in part by a grant from the Forschungskredit of the University of Zurich, Switzerland, and by internal research funds at CRS4, Italy. The authors wish to thank Paul Tafforeau for the help of acquiring the tooth dataset, which was scanned at the SLS in Switzerland (experiment number: 20080205).

References

[1] vmmlib: A vector and matrix math library. http://vmmlib.sf.net.

[2] B. W. Bader and T. G. Kolda. Algorithm 862: Matlab tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006.

[3] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz. Smooth mixed-resolution GPU volume rendering. In *Proc. IEEE/EG Symposium on Volume and Point-Based Graphics*, pages 163–170, 2008.

[4] T.-c. Chiueh, C.-k. Yang, T. He, H. Pfister, and A. E. Kaufman. Integrated volume compression and visualization. In *Proc. IEEE Visualization*, pages 329–336. Computer Society Press, 1997.

[5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. Symposium on Interactive 3D Graphics and Games*, pages 15–22. ACM SIGGRAPH, 2009.

[6] L. de Lathauwer, B. de Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal of Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[7] M. Do and M. Vetterli. Pyramidal directional filter banks and curvelets. In *Proc. IEEE Image Processing*, volume 3, pages 158–161, 2001.

[8] M. Do and M. Vetterli. The contourlet transform: an efficient directional multiresolution image representation. *IEEE Trans. Im. Proc.*, 14(12):2091–2106, 2005.

[9] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. AK Peters, 2006.

[10] N. Fout, H. Akiba, K.-L. Ma, A. Lefohn, and J. Kniss. High quality rendering of compressed volume data formats. In *Proceedings Eurographics*, pages 77–84, Jun 2005.

[11] N. Fout and K.-L. Ma. Transform coding for hardware-accelerated volume rendering. *IEEE Transaction on Visualization and Computer Graphics*, 13(6):1600–1607, 2007.

[12] E. Gobbetti, F. Marton, and J. A. I. Guitiàn. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, Jul 2008.

[13] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proc. IEEE Visualization*, pages 53–60, 2002.

[14] J. A. Iglesias Guitián, E. Gobbetti, and F. Marton. View-dependent exploration of massive volumetric models on large scale light field displays. *The Visual Computer*, 26(6–8):1037–1047, 2010.

[15] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *Siam Review*, 51(3):455–500, Sep 2009.

[16] P. Ljung, C. Lundstrom, and A. Ynnerman. Multiresolution interblock interpolation in direct volume rendering. In *Proc. Eurographics/IEEE TCVG Symposium on Visualization*, pages 259–266, 2006.

[17] P. Ljung, C. Winskog, A. Persson, C. Lundstrom, and A. Ynnerman. Full body virtual autopsies using a state-of-the-art volume rendering pipeline. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):869–876, Oct 2006.

[18] NVIDIA. *CUDA C best practices guide*, version 3.2 edition, Aug 2010.

[19] NVIDIA. *CUDA C programming guide*, version 3.2 edition, Nov 2010.

[20] R. Parys and G. Knittel. Giga-voxel rendering from compressed data on a display wall. In *WSCG*, 2009.

[21] F. Rodler. Wavelet based 3D compression with fast random access for very large volume data. In *Proc. Pacific Graphics*, pages 108–117, 1999.

[22] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proc. IEEE Visualization*, pages 293–300, 2003.

[23] S. K. Suter, C. P. Zollikofer, and R. Pajarola. Application of tensor approximation to multiscale volume feature representations. In *Proc. Vision, Modeling and Visualization*, pages 203–210, 2010.

[24] Y.-T. Tsai and Z.-C. Shih. All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. *ACM Transactions on Graphics*, 25(3):967–976, 2006.

[25] J. E. Vollrath, T. Schafhitzel, and T. Ertl. Employing complex GPU data structures for the interactive visualization of adaptive mesh refinement data. In *Proc. Volume Graphics*, pages 55–58, 2006.

[26] H. Wang, Q. Wu, L. Shi, Y. Yu, and N. Ahuja. Out-of-core tensor approximation of multi-dimensional matrices of visual data. *ACM Transactions on Graphics*, 24(3):527–535, Aug 2005.

[27] Q. Wu, T. Xia, C. Chen, H.-Y. S. Lin, H. Wang, and Y. Yu. Hierarchical tensor approximation of multidimensional visual data. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):186–199, Feb 2008.

[28] B.-L. Yeo and B. Liu. Volume rendering of DCT-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43, 1995.