Interactive Histology of Large-Scale Biomedical Image Stacks

Won-Ki Jeong, Jens Schneider, Stephen G. Turney, Beverly E. Faulkner-Jones, Dominik Meyer, Rüdiger Westermann, R. Clay Reid, Jeff Lichtman, and Hanspeter Pfister, *Senior Member, IEEE*

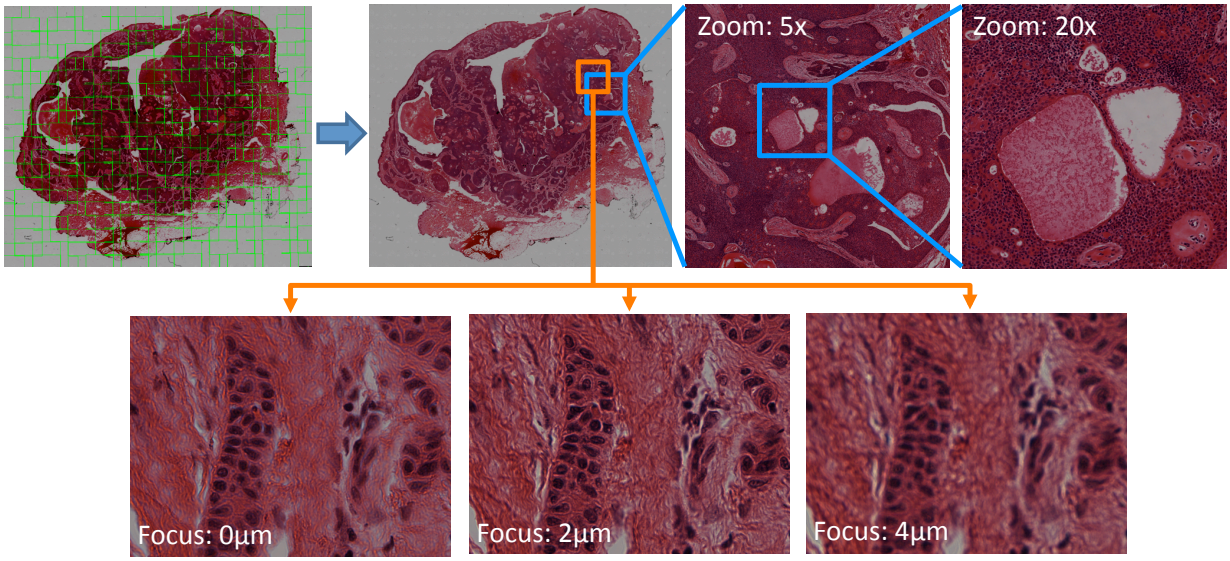

Fig. 1. Our display-aware gigapixel image viewer for biomedical image stacks. Input image stacks are processed and stored individually (green rectangles to the left, 360 image stacks each comprising 2.7K × 2K × 16 samples, resulting in over 30 gigapixels), but the globally consistent view for an arbitrary zoom level and image plane can be composed on-the-fly (right). The proposed system provides an interactive digital pathology workflow that allows fast changes in pan, zoom, and focus.

Abstract—Histology is the study of the structure of biological tissue using microscopy techniques. As digital imaging technology advances, high resolution microscopy of large tissue volumes is becoming feasible; however, new interactive tools are needed to explore and analyze the enormous datasets. In this paper we present a visualization framework that specifically targets interactive examination of arbitrarily large image stacks. Our framework is built upon two core techniques: display-aware processing and GPU-accelerated texture compression. With display-aware processing, only the currently visible image tiles are fetched and aligned on-the-fly, reducing memory bandwidth and minimizing the need for time-consuming global pre-processing. Our novel texture compression scheme for GPUs is tailored for quick browsing of image stacks. We evaluate the usability of our viewer for two histology applications: digital pathology and visualization of neural structure at nanoscale-resolution in serial electron micrographs.

Index Terms—Gigapixel viewer, biomedical image processing, GPU, texture compression.

1 Introduction

The use of digital imaging is widespread in basic biological research but is less common in histopathology. The practice of anatomic (surgical) pathology involves the use of a microscope to view tissue mounted on a glass slide, an approach that has not changed significantly over many decades. Pathologists have been slow to transition from glass slides to digital imaging for reasons that relate to performance and interpretation. High-resolution digital images are necessary for accurate diagnosis, and because of the size of histological sections, images have been too large to handle efficiently. A major obstacle to the use of digital imaging in pathology has been the inability to display large images at interactive rates.

Large microscopy images are generated in three steps: first, by acquiring tiled scans or images, and, second, by preprocessing to correct, align, and stitch the tiles into a seamless montage. The final step is to create a multiresolution image pyramid allowing efficient display of the image based on the current viewpoint and screen size. The preprocessing works well for moderate-sized images but not for extremely large images (e.g., terapixel images). Because the acquisition and processing steps are performed sequentially, there can be a long delay before a slide image is available for viewing. Many times, one may only be interested in viewing a small portion of the image dataset. In this case the effort to create globally consistent data is unwarranted. Finally, a globally consistent data structure is not well suited to dynamic

- *Won-Ki Jeong and Hanspeter Pfister are with Harvard University, E-mail: {wkjeong, pfister}@seas.harvard.edu.*
- *Jens Schneider is with King Abdullah University of Science and Technology (KAUST), E-mail: jens.schneider@kaust.edu.sa.*
- *Stephen G. Turney and Jeff Lichtman are with the Department of Molecular and Cellular Biology at Harvard University, E-mail:{sturney, jeff}@mcb.harvard.edu.*
- *Dominik Meyer and Rüdiger Westermann are with Techniche Universität München, E-mail: meyerd@in.tum.de, westermann@tum.de.*
- *Beverly E. Faulkner-Jones is with BIDMC Pathology and Harvard Medical School, E-mail: bfaulkne@bidmc.harvard.edu.*
- *R. Clay Reid is with the Department of Neurobiology at Harvard Medical School, E-mail: clay_reid@hms.harvard.edu.*

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

(local) data modification. Dependencies within the data structure make it necessary to recreate the entire image pyramid—a time-consuming process.

Because the major bottleneck of handling large images is slow disk access, reduction of the data size is essential to achieving interactive response in the image viewer. Most of the existing large image viewers rely on progressive transmission of image tiles and speculative prefetching to hide disk access latency. Those techniques can cover small local movements, but can be easily outpaced by rapid changes in pan, zoom, and focus. In a clinical setting, pathologists need to be able to inspect large high-resolution slide images quickly. Therefore, it is necessary to use a data compression scheme to minimize data transfer overhead.

In this paper, we introduce a novel visualization framework for interactive histology of extremely large microscopy image stacks. Our framework employs a local, adaptive data structure with demand-driven processing to handle extremely large images efficiently (see Figure 1 for a confocal microscopy data set comprising over 30 gigapixels). We call this *display-aware* processing. The core idea of the display-aware framework is to use *lazy, on-demand* evaluation when accessing the image data. Unlike existing methods [18, 10, 35], we do not manage a fully-processed, global image pyramid per image slice. Instead we keep track of high-level data dependency and local correspondence between the input image tiles. The globally-aligned view for an arbitrary viewpoint and zoom level is composed on-the-fly by fetching only a small subset of the input dataset and applying necessary computations whose cost is bound to the current display size. Our display-aware framework allows fast random access to the image data at any scale and does not require time-consuming preprocessing. For interactive performance, we propose a novel GPU 3D texture compression that exploits the similarity between successive slices to increase the compression ratio and allows realtime texture decompression on the GPU. Our 3D texture compression is specifically tailored for fast browsing across slices of pathology-slide image stacks. Compared to previous GPU-based compression methods, our method offers an unprecedented flexibility in the choice of available bitrates, thereby allowing a very fine quality vs. bitrate trade-off.

2 Overview of Histology

Histology has two main application areas: anatomic pathology and basic research. Anatomic or surgical pathology involves the histopathologic examination of tissues from biopsies or larger excision specimens. Tissues are typically fixed with formalin, dehydrated, and then infiltrated with paraffin wax prior to being sectioned at about four to five microns (μm). Sections are permanently mounted on glass slides and stained with hematoxylin and eosin (H&E). The pathologist views the sections using a microscope, usually at several different magnifications (e.g., 2×, 4×, 10×, 20× and 40×). The morphologic changes in the tissue are assessed and the morphologic and clinical information are then integrated to render the diagnosis. Precisely how the pathologist acquires the morphologic information from the slide varies between individual operators. Diagnostic information is obtained from both moving and static images, and how the slide is driven can impact the ability of the pathologist to render a diagnosis. An experienced pathologist can make a large number of diagnostic decisions each working day and the currently used glass slide-based process is fast and efficient (Figure 2).

Whole-slide digital imaging is receiving increased attention in surgical pathology but it is still not widely used in a diagnostic setting [16]. A challenge in digital pathology is to make image quality good enough to be comparable to the experience of using a microscope [11]. One aspect of improving image quality is to achieve high resolution and color fidelity. Another is to acquire multiple focal planes. Even if slides are digitized as high-resolution image stacks, an important and as of yet underdeveloped feature of slide viewer software is the ability to translate and change focus and magnification as quickly and seamlessly as when viewing slides on a microscope [21]. While automated whole-slide imaging systems are becoming fast (e.g., digitizing a single 20mm×20mm section at 20× magnification in 1 minute per image plane), viewer software is currently designed more to handle the difficult task of accessing huge amounts of slide image data (one or more gigapixels for a single two-dimensional slide image) efficiently than the perhaps equally challenging task of making display of slide image data effective for the human visual system [20]. Existing implementations offer the ability to translate and zoom a large two-dimensional image or to step through an image stack, but none so far offer the ability to do all three (translate, zoom, and change image plane) with minimal delays. The rapid flow of visual information captured by the eye when using a microscope provides information that is used in diagnostic decision making. For this reason, image latency is a major obstacle to the usefulness of digital methods in pathology. To gain acceptance, the slide viewer software needs to mimic the performance of a microscope and allow for individual preference with respect to viewing dynamic and static images.

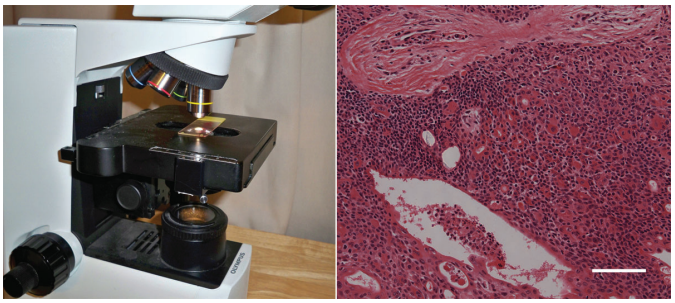

Fig. 2. Conventional light microscope on which a pathologist views glass slides moved by hand. An example H&E-stained histologic section imaged at 20× objective magnification with a resolution of 0.26 μm per pixel. For a 16mm × 10mm section, the size of the composite two-dimensional image is approximately 2.2 gigapixels. Scale bar is 100 μm.

In basic research, the tissue on a slide is often labeled with one or more fluorescent markers. The fluorescence can be imaged in thin optical sections (fractions of a micron thick) using confocal microscopy. With multiple channels, the size of a single image stack can easily exceed one gigabyte. At high resolution, the field of view may be less than 100 μm on a side. Thus imaging a 20mm×20mm region would require the acquisition of thousands of image stacks. Even for small data sets comprising tens of image stacks, a major challenge is the ability to align, stitch, and access montages of image stacks.

Our display-aware framework facilitates the viewing of arbitrarily large three-dimensional image volumes. We avoid the need for global alignment and stitching of image stacks as a preprocessing step. This virtually eliminates the delay between acquisition and analysis, since only the alignments necessary for the current view are computed. And due to our GPU-friendly texture compression, image sizes and bandwidth requirements are considerably reduced.

3 Previous Work

There are several research papers and commercial software to display large-scale images interactively. Kopf et al. [18] proposed the methods for acquisition, processing, and display of panoramic images, which was later released as Microsoft HDview [27]. Several web-based high-resolution panoramic image viewers were also proposed [10, 35]. Commercial virtual microscope software extends panoramic image viewers to display large microscopic images [17, 29]. Most existing large-scale image viewers rely on a global hierarchical data structure for fast and efficient data management, which is not scalable to a workflow that is required to handle an arbitrarily large collection of images. In addition, none of those method addresses how to quickly pan, zoom, and switch between multiple high resolution images.

It is clear from these examples that the foremost goal of compressing such images is always a reduction in data size. This reduction makes data more tractable in general, allowing pathologists to carry with them, transmit, and visualize larger amounts of data without ad-

ditional hardware costs. Especially in histology, where lossy compression is widely accepted as means to reasonably deal with the overwhelming amounts of data, the potential for storage requirement reductions are considerable. As there is an abundant body of work on data compression, we refer to standard references [32, 12] and review only work closely related to ours.

The goal of data compression has been recognized in the visualization community and it has spawned a trend to move from CPU-based compression [38, 13, 14, 30, 39] to GPU-based compression. The major reason is that the GPU's internal bandwidth (currently exceeding 140GB/s) is vastly superior to the PCIe's theoretical bandwidth of 4GB/s. However, the step from CPU-based to GPU-based decoding implies the usage of compression algorithms that can be decoded in parallel. Since this is easier to achieve for lossy compression, the literature on this topic is decidedly richer than on lossless compression in the context of GPU-based coding.

Among the first authors to propose GPU-based decoding were Lum and Ma [25]. They propose to compress time varying volume data by grouping four timesteps of a scalar volume to form a volume of four-dimensional vectors. Assuming high temporal coherence, the information stored in these vectors is then compacted using a four-tap DCT and each component is quantized using Lloyd's scalar quantizer [24]. The decoding then relies on paletted textures, an early form of dependent texture lookups. The method can decode volumes at high speed, which can be mostly attributed to the fact that only fixed bitrates are used.

Based on the concepts of Laplace pyramids [4, 9] and vector quantization [30, 8], Schneider and Westermann [34] generalize a two-dimensional framework by Beers et al. [1] and propose a hierarchical encoding of scalar-valued volumetric data [34] as well as its on-the-fly decoding in a fragment shader. The key idea is to predict details in the volume by a coarser, subfiltered version of the volume. The difference between prediction and actual volume is then grouped into 2^3 vectors, properly normalized (mean removal and scaling components to the range $[0, 1]$), and quantized. Fout et al. [6] improve the method to address the coding of time-varying and multivariate volume data. Fout and Ma [5] also describe a method that uses transform coding followed by classified vector quantization to achieve higher fidelity at more flexible bitrates. Wang et al. [37] describe a compression method for partitioned volume data residing in an octree. They also address texture packing in order to achieve a wider range of possible bitrates. The texture format provided by the graphics API that fits the desired output bitrate closest is automatically chosen and each code word is padded to this format. They report that using this strategy the padding overhead is below 10%. In contrast, we utilize buffer textures, a concept introduced recently to the OpenGL rendering API, to avoid any padding except for reasons of convenience.

Our approach to compression is very similar in style to the aforementioned approaches. However, the data we are concerned with will always show an extremely high coherence between slices (confocal planes) of the image. Consequently, the key idea is to encode the frontmost and backmost slices at a rather high bitrate using hierarchical vector quantization. Then, interior slices are predicted using information from the front- and backmost slice, and only the differences to this prediction are encoded. Such predictor/corrector methods have been shown to be extremely successful in the context of time-varying volumes [36, 26]. Unlike previous approaches, we offer a flexible bitrate control. The only limitation is that the bitrate of each quantizer stage is constant per vector, but as detailed in Section 6 this can be any rate.

4 Definitions and System Overview

We define an *image slice* as a collection of pixels imaged on the same focal plane. An *image stack* is a collection of image slices that are created by scanning the same spatial region on different focal planes. Figure 3 left describes the image slice and image stack pictorially. Because the field-of-view of a high magnification objective is much smaller than the entire imaging region, we collect a set of image stacks that overlap about 5–10% at their boundary. Each individual image stack covers only a small subregion.

We define the *scale* in our context as the distance between the viewer and the object—or equivalently the size of the field of view—in the global coordinate system (reference space in Figure 3) while *resolution* refers to the number of pixels that defines the actual size of an image. Therefore, scale is the measure of how close the object is to the viewer and how much detail can be seen by the viewer, which corresponds to the magnification factor of the microscope objective.

The input to our system is a large collection of image stacks, e.g., hundreds to thousands of high-resolution images, up to terabytes of raw data size in total, for a few square centimeters of the biological sample. Each image stack usually consists of 9 to16 image slices for a tissue sample of about five microns thick, and the distance between adjacent slices is about one micron.

The first step in our system is building image stack pyramid. For each image stack, we create coarser resolution image stacks (Section 5.3, Data Structure). Then each image stack is aligned in a reference coordinate system (Figure 3 middle, Reference Space) and per-stack geometric transformation is stored in Data Space. In the mean time, each image stack pyramid is diced into pre-defined sized sub-stacks, and those stacks are compressed using our compression scheme (Section 6). A spatial data structure for sub-stacks, e.g., the bounding box hierarchy, is also built to search visible stacks quickly. Finally, our display-aware viewer fetches only currently visible image stacks from disk or cache (Section 5) and decompress them on-the-fly using the GPU to display the current image on the screen (Section 6.5). The details of the two main components of our system, *display-aware framework* and *texture compression*, will be introduced in the following sections.

5 Display-aware image viewer

Our display-aware image viewer framework provides an efficient method to process only the visible portion of image stacks on the fly without creating a global view of the data in advance. In addition, we propose an efficient adaptive image hierarchy technique to reduce the data size without sacrificing the image quality (Section 5.2). An important goal of our display-aware framework is that the original, measured data is never altered but only augmented with additional information. This additional information includes the proposed compression scheme. This has two benefits. Firstly, measuring data is a time-consuming and expensive process. Hence the original data should be retained for future algorithmic advances. Secondly, while not noticeable in typical scenarios, our compression scheme is lossy and might introduce artifacts. Consequently, we use the compressed representation of the data to achieve interactive updates, while the original data can be presented to the user if her or his browsing behavior permits to do so.

5.1 Description of the Display-Aware Framework

The core idea behind our display-aware framework is to separate the raw data space and the display space. Our framework consists of three conceptual spaces: *data*, *reference*, and *display* space. We now formally define the workflow and data structure we propose.

Data Space This is the space where the input raw image stacks and per-stack scale and geometric transformations are stored (Figure 3 left). Each image stack is independently stored in a multiresolution format. Only the image chunk with a proper resolution for a given scale will be used to compose the final image in the display space. The per-stack scale defines the scale of that stack in reference space. The per-stack geometric transformation defines the mapping between the local coordinate system in data space to the global coordinate system in reference space. There is no single global pyramid that contains the whole image as a single stack. Instead, each small image stack is converted into a small local pyramid. By doing this, preprocessing time is greatly reduced, and the depth of the pyramid hierarchy is much more shallow, which allows faster memory access. Because each slice is naturally decomposed into small pieces, random access to a local region can be done simply by fetching the corresponding image stacks without reconstructing the pyramid in a coarse-to-fine manner, similar

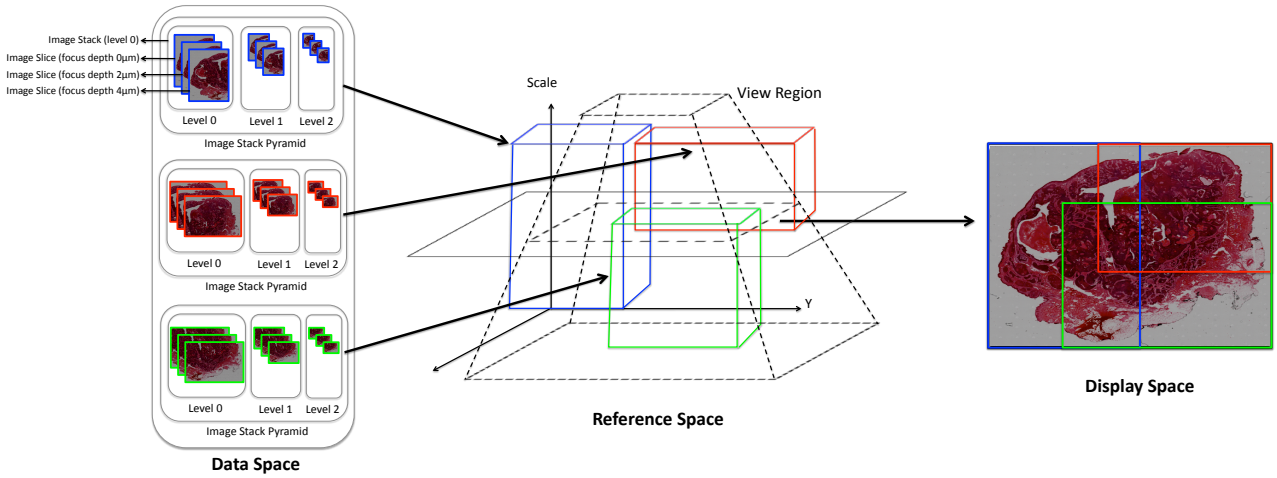

Fig. 3. Pictorial description of the proposed display-aware framework. Left: each Image stack pyramid consists of image stacks in different scales (level), and each image stack consists of image slices imaged on different focal planes. Middle: a single image slice (not an image stack) is mapped to a 3D cube in reference space. The top face of a cube is mapped to the image slice in the highest scale (e.g.,level 0), while the bottom face is mapped to the image in the lowest scale (e.g., level 2). Right: Final image on the screen is composited on-the-fly using the image slices that intersect with the current viewing window.

to existing methods. This difference becomes more relevant in pathology because moving across different slices is one of the most common tasks in pathology. If a pathologist zooms in to the highest resolution and moves to the next slices then existing methods must reconstruct all dependent levels, which is very expensive. On the other hand, our display-aware framework can fetch the data only for the current screen size, and there is no overhead due to the data dependency.

Reference Space Reference space is a three-dimensional global coordinate space to which all the images are mapped (Figure 3 middle). In data space, each image stack is treated independently and we do not have a global view of them. Thus, the main purpose of using reference space is to construct a spatial and scale relationship between image stacks in a common coordinate system. In reference space, the x- and y-axis represent the spatial location, and the z-axis represents the image scale. Therefore, an image stack in reference space is a set of three-dimensional, axis-aligned bounding boxes (one per each slice in the stack) for which the x-y location is computed by a corresponding geometric transformation. The current viewing window is a two-dimensional rectangle in the x-y plane in this space. Efficient spatial data structures can be used to quickly find the visible images for any given location and scale in this space. This space is purely a virtual, continuous space without any concept of actual pixel values. Sampling will be done in display space.

Display Space Display space is a local coordinate space where the pixel values are defined and actual image operations take place (Figure 3 right). For any given region of interest and scale, the image slices within the region in the reference space are resampled and any per-slice operations, such as color adjustment or image alignment, are performed. Once the resampling and computation are done, we only store per-slice color and transformation parameters in data space (Figure 3 left) and never modify the actual pixel values of the raw input image slices.

5.2 Adaptive Image Hierarchy

Adaptive refinement techniques have been widely used in many different fields, such as numerical simulation [3] and computer graphics [7, 2], to reduce computational cost and memory footprint. The main idea of these approaches is to refine a coarse grid to achieve a nested grid that offers higher resolution where higher accuracy is required. Most existing methods focus on building a global nested hierarchy, such as adaptive refinement of an octree.

We can employ a similar idea to display the data samples using spatially varying sampling resolutions in order to reduce the data size

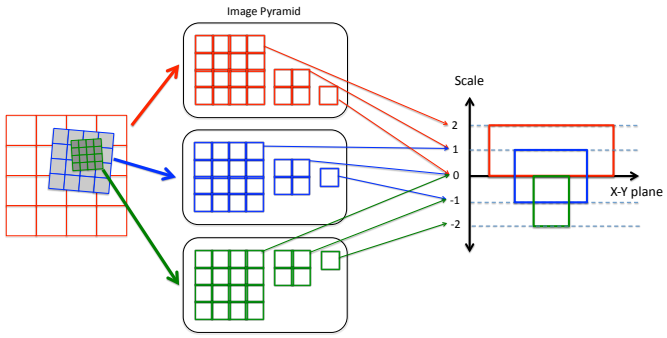

Fig. 4. Adaptive image hierarchy. Each nested image represents a region of twice higher magnification than its outer image in display space (left). Note that the images are neither resampled into a global data structure nor aligned to a common coordinate system. Each image has its local image pyramid in data space (middle). Each pyramid level (local to the image) is mapped to a global scale value in reference space, and images with different scales are shifted along the scale axis accordingly (right).

while at the same time allowing deep zooming into the region of interest. Instead of creating a full scan of the entire region at the highest magnification, we can create multiple images at different spatial locations and scales, i.e., using more pixels for interesting regions, and compose them into a single coordinate space. Unlike other methods based on a global image pyramid, our method can naturally handle multiple nested images without resampling or updating the data structure. This allows us to easily create a deep zoom hierarchy. Even more, images with arbitrary orientation and scales can be combined easily. We only need to store a per-image scale and geometric transformation to align them in reference space. Figures 4 and 5 depict the adaptive image hierarchy of three images with different scales.

Because each image is scanned at a different scale and location, they have to be aligned using an image registration method. We can assume that the microscope stage movement is translational, so we only need to find x and y displacements from one image to the other. We use a GPU implementation of rigid image registration to interactively align images [15]. The image scale is proportional to the magnification of the microscope objective.

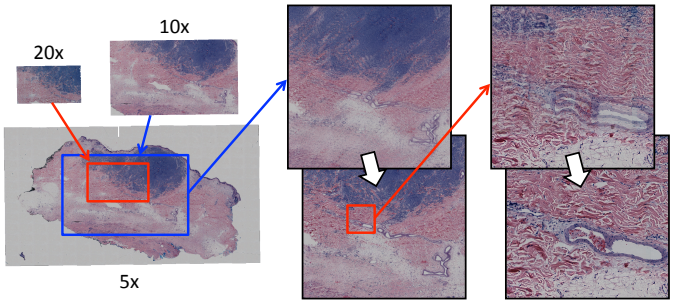

Fig. 5. Building an adaptive image hierarchy using on-the-fly GPU image registration. Three images with varying magnification are aligned in the common coordinate space. A coarse initial location of each image is manually given by the user, then the GPU rigid registration aligns two consecutive images quickly (see the supplementary video). 10× and 20× images are aligned to the background image, respectively (right).

5.3 Implementation Details

Data Structure For each input image stack, a multiresolution pyramid is constructed. Each slice in the image stack is independently reduced for coarse levels, and only x and y dimensions are halved. Then each level in the pyramid is diced into fixed-size 512 × 512 × (number of slices) tiles. Because the image size is not always a multiple of 512, the regions outside the image will be padded with zeros. Each tile keeps the size of the valid region so that the renderer can skip displaying the padded region. Tiles can be directly accessed from the pyramid using a level and tile index. For each render pass, the corresponding image level in the pyramid is chosen based on the current viewing scale. Each image stack is a pyramid of image stack tiles, and a slice is a collection of image stack pyramids. We construct a pyramid with tile pointers where most of the pointers are null at the beginning. As the user navigates the image, the actual tile data dynamically flows in and out from the tile cache system.

For the adaptive image hierarchy, each image stack is given a scale value to determine the relative image size and level in reference space. We choose one base image stack for scale 0, and assign relative scale values to other image stacks so that a scale value n represents $2^n\times$ magnification from the base image. For example, an image of scale 1 corresponds to a $2\times$ magnification of an image of scale 0. When the images are composed in reference space, the internal pyramid level is also shifted by the image scale. For example, a pyramid level 0 in the image of scale 0 is same level as a pyramid level 1 in the image of scale 1. Figure 4 depicts mapping of the pyramid levels and scale in reference space.

Software Cache and Prefetching Schemes Each tile dynamically allocates memory space for the CPU and GPU. CPU memory is used to store the compressed stack data loaded from the disk. GPU memory is used to store a GL buffer object for the compressed stack data from CPU memory and a 512 × 512 2D OpenGL texture to store and display the decoded slice. A two-level software cache system manages buffering data between disk, the CPU, and the GPU. A LRU (least-recently-used) scheme is used to free in-cache tiles when the cache is full.

We observed that the major bottleneck in our system is the disk access time. Loading a compressed 512 × 512 × 16 stack of about 600 KBytes in size from a SATA hard drive usually takes 20–50 milliseconds, and, occasionally, it reaches up to a hundred milliseconds. We minimize the disk latency by prefetching neighboring tiles in the background or during idling while a concurrent display thread fetches tiles which are currently visible in real time. A cylindrical region that spans several scales and has a twice-as-large spatial extent on the current scale as the current view port is prefetched for zooming and panning motions. In the rare case that a prefetching process cannot be fully completed since the user moved the viewpoint too quickly, the current prefetching is immediately terminated and a new prefetching process is started for the new viewport.

6 Texture Compression

Our display-aware framework provides an efficient data management scheme, but to be able to transfer image stacks from the disk or SSD to the GPU without lags we need to further reduce the size of each stack by using a GPU-based texture compression scheme. More specifically, we use a novel texture compression approach that is based on predictive, hierarchical vector quantization. It is custom tailored to the type of data we are dealing with, i.e., data exhibiting exceptionally high coherence in between the slices of each image stack. Our approach offers faster encoding than the industry standard S3TC, while at the same time achieving higher fidelity and higher compression ratios. At a glance, the method works as follows:

1. Group input data into stacks of N image slices, each of which comprises 512×512 pixels.
2. For each stack, encode the first and last slice jointly.
3. Perform a linear interpolation to predict the $N-2$ intermediate slices and encode the difference between the actual and the predicted values.

6.1 Vector Quantization

Although a full discussion of vector quantization is beyond the scope of this paper, we will briefly review the basic underlying concepts in this section. For a more complete discussion, we refer the reader to Neuhoff and Gray's excellent survey [12].

Given a set $\{x_i\}_{i=1}^N \subset \mathbb{R}^d$ of d-dimensional vectors. A vector quantizer replaces this set by an approximation consisting of an ordered set of indices, $\{\alpha_i\}_{i=1}^N \subset \mathbb{N}$, and a ordered set of *codebook* vectors, $\{c_i\}^n \subset \mathbb{R}^d$. The original set of vectors can then be reconstructed by the decoder by performing a simple lookup into the codebook, i.e., $\tilde{x}_i = c_{\alpha_i}$. Since the process is typically lossy, a error metric can be defined to assess the quality of the encoding/decoding round-trip, $\xi(x_i) = (x_i - \tilde{x}_i)^T(x_i - \tilde{x}_i)$. This basic setting leaves implementations with two tasks. Firstly, an appropriate codebook has to be found, and secondly, the best index set with respect to the error metric ξ has to be found. Once an initial codebook is specified, the Linde-Buzo-Gray (LBG) algorithm [22], also known as Generalized Lloyd algorithm (GLA) [24] can be used to refine both the index set as well as the codebook. Compression is achieved if the number of bits necessary to represent the codebook and the index set is lower than the number of bits needed to represent the original set of vectors. The input vectors are typically assumed to be in some form of floating point format, in our case at 32 bits per component. The codebook can be further compressed by choosing a more compact representation. In our implementation, we chose a fixed-point format at a lower bitrate, since we seek to represent 8 bit color values.

Our implementation is very similar to [34] for which additional details are described in [33]. This particular implementation forms an initial codebook by maintaining a prioritized list of quantization cells. In this context, the quantization cells are a partition of the set of input vectors, where all vectors in a cell share the same index. Starting with only one bin containing all vectors, the quantization bin with the largest error is split into two new bins using principal component analysis (PCA) to determine an optimal split plane. Once certain threshold criteria are met, the generation of the initial codebook is stopped and the codebook thus obtained is refined using the LBG algorithm.

To further increase the efficiency, we split each image in a series of frequency bands, where low-frequency bands show less detail and can therefore be encoded using less bits. To propagate contributions from these low-frequency bands to higher frequencies, we use a smooth filter for upscaling. This gives a low-frequency prediction of the full image that can be corrected by using vector quantization on the details yet missing at this particular stage.

In the following subsections we will discuss our particular compression scheme along with design decisions.

6.2 Joint-Coding of Front and Back Slice

We start by *reducing* the frontmost and backmost slice of the stack to 2×2 pixels. This is accomplished by computing the average over four blocks of 256×256 pixels each and is equivalent to a recursive application of a 2×2 box filter followed by subsampling (see also Figure 6). The resulting four colors per slice are then encoded in $2 \times 2 \times 2 \times 16$ bits in R5G6B5 format.

For each of the two slices, we then *expand* these 2×2 pixel images to 32×32 pixels each. As depicted in Figure 7, expansion proceeds by recursive bilinear upscaling using the C^1-continuous synthesis filter described in [19]. This doubles the image resolution along each axis in each step. The advantage of recursive upscaling vs. one-step bilinear upscaling is that the resulting prediction is smoother and free of bilinear interpolation artifacts.

Fig. 6. Recursive reduction step. The coarser level (light red) is generated by application of a 2×2 box filter to the finer level (light blue) followed by subsampling.

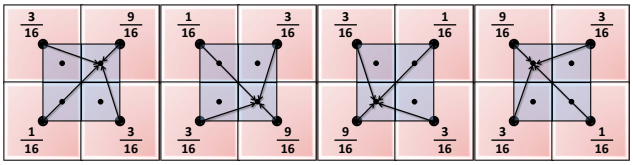

Fig. 7. Recursive expansion step. The finer level (light blue) is interpolated from the coarser level (light red) using the C^1 filter from [19].

The prediction is then compared to the input slices reduced to 32×32 pixels each. The difference between reduced input slices and the prediction are then grouped to form $2 \times 2 \times \mathrm{RGB}_f\mathrm{RGB}_b$ vectors, where the subscripts f and b denote the respective pixel of front and back slice. This difference is then encoded using the vector quantizer described in [34]. The vector quantizer replaces the $32 \times 32 \times 2$ RGB values with 16×16 $n-$bit indices into a codebook storing 2^n 24-dimensional vectors. Since the entries of the codebook are generally produced at floating point precision, we first clamp them to $[-63,+64]$ before quantizing them uniformly to 5 bits per component.

The reconstructed differences are then added back to the prediction and upscaled two more times, once to 128×128 and once to 512×512. In the first case, 5 bits per codebook component are used, whereas the last stage clamps differences to $[-63,+64]$ and quantizes them to 8 bits per component.

The reason for the clamp / quantization process on the codebooks is that we observed that most differences are already in the interval $[-63,+64]$ and that a significant number of bits can be saved in this manner. Note that we avoid error propagation by reconstructing the image from the encoding of the current stage before proceeding to the next stage.

6.3 Predicted Encoding of Intermediate Slices

Since intermediate slices vary only slowly, they can be predicted very well using a linear interpolation between the front and back slices. The best way to encode these intermediate values would be to compute the difference to the prediction, then use a $(N-2)\times$RGB-dimensional vector quantizer to store these differences. However, this results in an encoding time that is about twice as high as our final solution. Our current implementation is based on two observations.

1. Quantizing twice the amount of $(N-2)/2\times$RGB vectors is almost twice as fast in our optimized implementation.

2. The difference between prediction and actual data is largest in the middle, i.e., farthest away of the front and back slices.

Consequently, we split each $(N-2) \times 3$-dimensional vector into two $(N-2)/2 \times 3$-dimensional vectors denoted by u and v. u thus spans slices 1 through and including $(N-2)/2$, while v spans slices $(N-2)/2+1$ through and including $N-1$. In order to ensure that these two different vector sets exhibit similar data distributions over their components (i.e., large values at large indices), we reverse the component order of vector v. Even though the two slabs of slices are now encoded using the same codebook, this method results in a notable color difference between slices $(N-2)/2$ and $(N-2)/2+1$. This artifact can be reduced significantly by extending the distance function to a weighted distance function between vectors,

$$\Delta(x,y) := \sum_i (x_i - y_i)^2 \, \omega_i^2, \tag{1}$$

where $x,y \in [0,\ldots,255]^2$ and ω_i is the respective weight. The ω_i may thus be used to ensure additional sensitivity at the 'end' of each vector. In our tests, we set the weights for slices $(N-2)/2$ and $(N-2)/2+1$ to 1.4 and the weights for slices $(N-2)/2-1$ and $(N-2)/2+2$ to 1.1. All other weights are set to 1. These values were found empirically and completely remove the aforementioned artifact for all of our test cases. Note that using a weighted distance function does not introduce an overhead to the decoding step since the weights are removed from the codebook at the end of the encoding phase. This stage truncates the components of the codebook to $[-63,+64]$ and stores them in 8 bits each.

A minor drawback of this method is that it slightly decreases the compression ratio since now twice the amount of indices have to be generated. Still, we show in the results section that sufficient compression rates at reasonable fidelity can be achieved. Inherently this description assumed N to be even. If this is not the case, we include the backmost slice in v, thereby actually encoding $N-1$ interior slices instead of $N-2$.

Since padding tiles that do not comprise 512×512 pixels with zeros introduces sharp boundaries, we provide special treatment for this case. This is especially important due to the fact that the predictor step will otherwise leak the padding into the data area, thereby compromising the achieved compression ratio. We therefore store the rectangle covering the valid data explicitly with the encoded stack and replace the color in the padded area with the slice average. While there exist better ways to resolve this issue, e.g., padding with local averages or pixels from adjacent stacks, this minor modification is fast and resolves most of this issue for our test cases.

6.4 Automatic Bitrate Control

The bitrate can be chosen separately for each quantization step. Since our vector quantizer starts with a single quantization bin and recursively splits one bin into two, we simply stop refinement when a certain error threshold is met. We measure this threshold in terms of the root-mean-squares error, rmse. For images that have a large homogeneous area and a small area with highly varying detail, however, this simple thresholding is unfit. The reason is that the large homogeneous area will guarantee a very low rmse although there might be very large errors in the small, detailed area. Thus, we introduce a second threshold that is based on an estimate of the maximum error. Providing a precise maximum error is computationally expensive, since it has to be completely recomputed after each recursive split, while the rmse can be updated incrementally. Therefore, we estimate the maximum

error after each bin split as

$$\xi_{\max} \approx \max\left(\xi_{\max}(\text{bin}_1), \xi_{\max}(\text{bin}_2)\right) \frac{\text{rmse}_{\text{old}}}{\text{rmse}_{\text{new}}}. \quad (2)$$

This heuristic tries to estimate the maximum error in the remaining, yet unsplit bins. It ensures that the codebook for image parts with large, homogeneous regions is refined although its average error is below the user-specified threshold. If the quantization bin responsible for the maximum error also has the highest average error, the split location and at least one of the new centroids are heavily dominated by the vectors responsible for the maximum error. In this case, our estimate tends to over-estimate the true maximum error and therefore results in a lower compression ratio than otherwise achieved, but ensures high image fidelity. Note, however, that if the bin to be split does not contain the maximum error, we might substantially underestimate the maximum error. Still, this estimate works well in practice and leads to a substantial improvement in image quality for those parts that are largely homogeneous and contain only a few, highly varying sections. Our conjecture is that the average error does in fact not decrease significantly until the quantization bin containing the outliers is actually reached. While a rigorous mathematical examination of this conjecture is beyond the scope of this paper, it is worth investigating in the future.

If *both* of the user-specified thresholds are met, i.e., the maximum and average error thresholds, we try to fill up available codebook slots until the number of bins reaches the next power of two. The rationale is to maximize the benefit of the index bitrate. If this is not possible, i.e., because we already represent the original data exactly, we append 0s to the codebook until we reach the next power of two. This keeps address calculations during decoding simple and thus ensures a faster decoding step. Additionally, we limit the amount of quantization bins for each of the four vector quantizers (32×32, 128×128, 512×512, and one for the intermediate slices). Further details and rate-distortion diagrams are provided in Section 7. Note that due to the 16-fold increase of resolution between levels the total amount of information per level increases. Consequently, we also increase the number of bins per level. Figure 8 shows a some examples of an image stack compressed at various bitrates. Recently, hybrid methods gained some attention [28]. Typically they start with a form of GPU-friendly encoding and use the LZO library [31] as a CPU-sided companion encoder to further increase the compression ratio. In our case, LZO compression resulted in less than 10% increase of compression ratio, yet it added additional latencies, so we chose to not pursue this direction further.

6.5 GPU-Based Decoding

Recent GPUs support bit arithmetic, hence the described compression method can be decoded on a GPU almost in a straightforward way. However, bitrates arising in our method are not a multiple of 8 and they can vary from stack to stack and quantizer stage to quantizer stage. The answer to this problem has traditionally been to pad entries to bitsizes provided by texture formats [28]. By storing the compressed data in buffers instead of textures, we are no longer bound by this restriction. The trade-off is that address computations have to be done in the shader code, but the gain in compression ratio is typically worth this minor drawback. We therefore store indices tightly packed followed by the codebook, again tightly packed, for each level. Between outputs of different quantizers, we pad to the next multiple-of-8-bit address and store this entry point to the next quantizer output to speed up address computations. These entry points are then directly fed to the respective shader.

Reconstruction then reverses the encoding process. Recursive expansion is performed by storing the lower resolution image in the appropriate level of a mipmapped texture. Computing the next finer level utilizes bilinear interpolation hardware and writes to another mipmapped render target. After the proper expansion has been performed, we simply fetch indices and codebook entries from the buffer and add the decoded details. Since we chose to organize the GPU buffers with 16 Bit granularity (GL_R16UI format), fetching a block of up to $k \times 16$ consecutive bits thus may require up to $k+1$ 16 bit

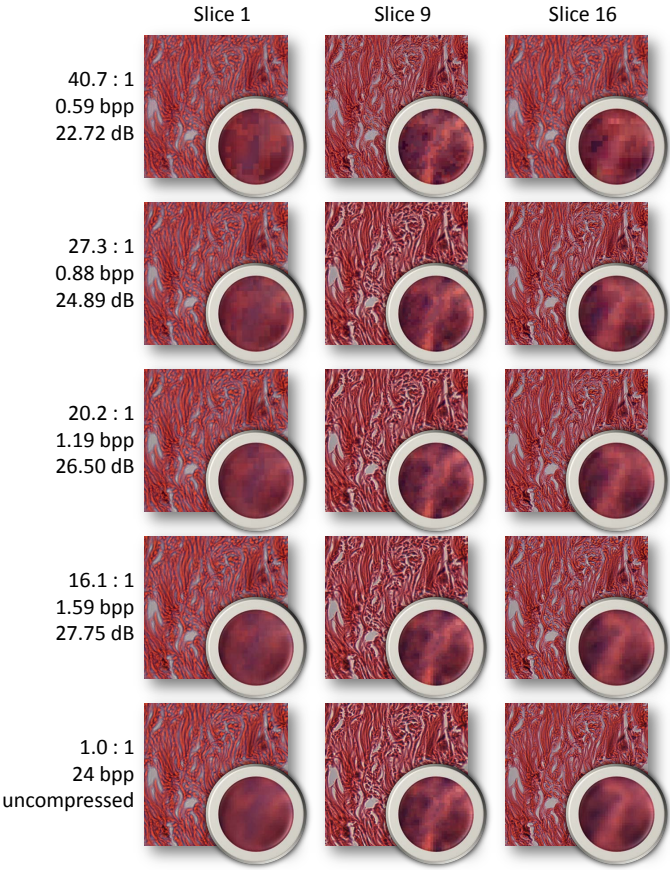

Fig. 8. Visual fidelity at various bitrates. In each row, three slices at a constant bitrate are depicted. In addition, a $10\times$ nearest neighbor zoom is shown for each image to demonstrate the high fidelity of our compression even at compression ratios between 16:1 and 20:1.

memory accesses. With respect to decoding speed, this appears to be a reasonable trade-off between lower granularity (less fetches, more data per fetch) and higher granularity (more fetches, less data per fetch). If the tile was padded, we clear the output texture with the desired background color and update only the valid data rectangle in the last update step.

If only the first or last slice is to be decoded, the decoder does not need to touch the data associated with the intermediate slices. If the user desires to see an intermediate slice, both front and back slice have to be decoded in addition to the differences encoded for the desired slice. However, since all operations involved in the decoding step are linear, we re-order these steps to speed up decoding.

First, linear interpolation along the stack's depth is performed before recursive upscaling takes place, thereby eliminating the need to expand two images. Secondly, since front and back slice are encoded jointly, the codebooks contain information on the front and the back slice at the same index. Consequently, it suffices to fetch the codevector and interpolate linearly between the respective components. There is no need to ever reconstruct more than just the slice the user wishes to see. Still, as detailed in Section 7, reconstructing intermediate slices is somewhat more costly than reconstructing the front or back slice.

The storage layout is schematically depicted in Figure 9. The figure also displays an exemplary data set at intermediate stages of the decoder.

7 Results

7.1 Decoder Performance

Our decoder is implemented in OpenGL and uses hardware accelerated bilinear fetches for the expansion filter. Decoding of a single front

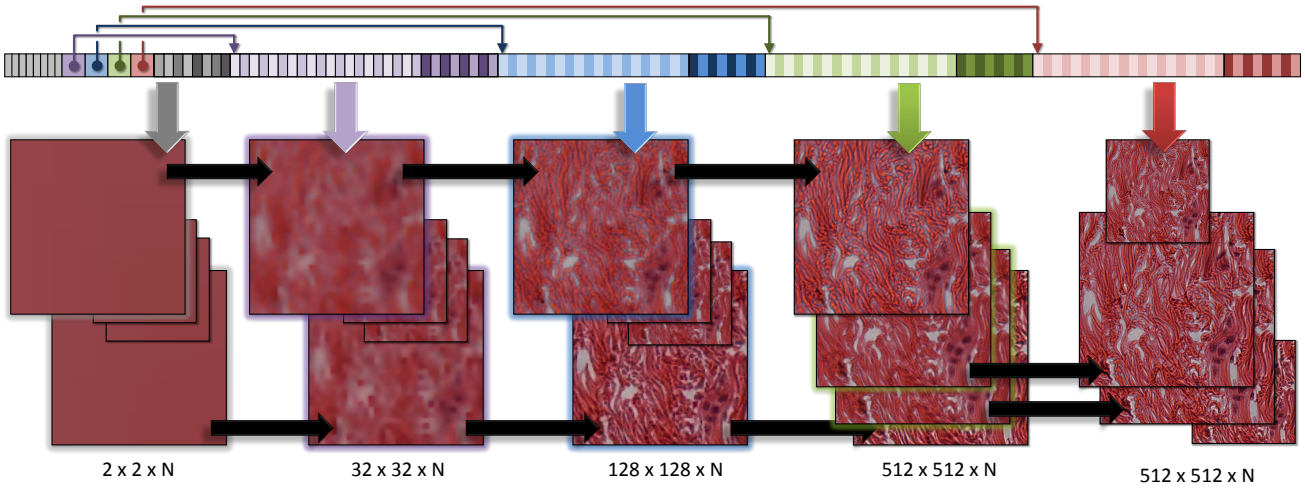

Fig. 9. Storage Layout of our decoder and exemplary data set at intermediate decoder stages(here, each slice was expanded to 512×512). After some header information, pointers to three of the four vector quantizer outputs are stored. The output of each quantizer is split into indices (light) and codebook (dark) and it is colored in a consistent hue. The first four stages decode the front- and backmost slice. The fifth stage adds details to the intermediate slices.

or back slice takes about 0.55ms on an NVIDIA GeForce 285GTX with 1 gigabyte dedicated video RAM. Decoding an interior slice takes about 0.73ms. This corresponds to a decoding speed of 454M pixel/s for front and back slices and 342M pixel/s for interior slices. The decoding speed is sufficient to power two full HD resolution displays or one 4M pixel display at more than 80 frames per second. The time provided here does not take into account the ability of our framework to cache decompressed images in order to avoid frequent decoding. The advantage of the proposed encoding method is its high compression ratio of more than 20:1 while at the same time maintaining a SNR of more than 26.5dB. The high compression ratio is mainly achieved by the fifth encoding stage that exploits the high coherence between slices and the C^1-continuous expansion between coding stages. However, it should be noted that this expansion consumes about 0.17ms per slice or up to about 30% of the total decoding time. This is mainly due to the fact that the implementation has to resort to ping-pong rendering since reading one mipmap level while at the same time writing at another one is not supported in OpenGL.

Encoding the first and last slices jointly is mainly motivated by the fact that we can predict intermediate slices by interpolation, as opposed to extrapolation if other slices were used. Also, for our data, all slices are reasonably focused and the inter-slice distance is very small. For data that does not have this property, our compression approach can still be used, but the bitrate to correct intermediate slices might be slightly increased.

We have compared our compression scheme to the JPEG standard and we generally observe that our compressor yields an SNR that is between 2dB to 3dB better than JPEG. However, we measured the SNR in both cases directly and in RGB space, which does not do the perceptual metric used for JPEG encoding justice. While images encoded with our compressor and JPEG looked very similar, some mild ringing artifacts in the JPEG version were detected while our compressor tends to show blocking artifacts at low bitrates. The major advantage of our compression scheme, however, is that it can be decoded on the GPU, which is involved at least for JPEG, due to the entropy coder used as a back-end in JPEG.

Figure 10 shows rate-distortion diagrams for one of our test data sets comprising 16 slices. Color values are assumed to be in the interval $[0,255]$. At 1.15bpp, we achieve an overall root-mean-squares error (rmse) of 7.34 (averaged over all 16 slices). The slice with the lowest distortion has an rmse of 5.73 and the one with the highest distortion has an rmse 9.17. This corresponds to SNRs of 24.63dB, 26.57dB, and 28.88dB respectively. Encoding time for one stack ($512 \times 512 \times 16\times$ RGB = 12MB of data) on a single core of a Xeon

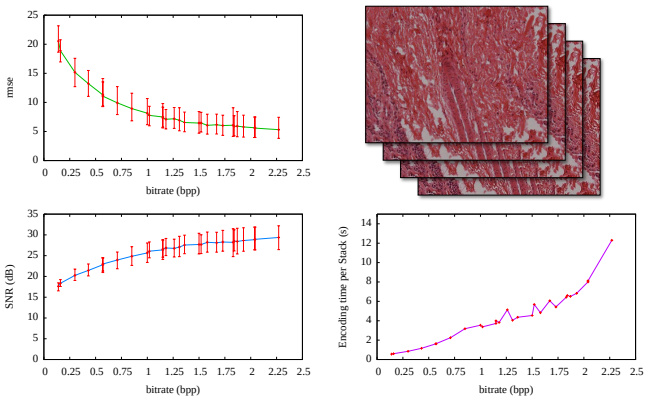

Fig. 10. Rate-distortion diagrams for a typical confocal microscopy data set. Top left: root-mean-squares error over bitrate. Top right: representative part of the data set. Bottom left: SNR over bitrate. Bottom right: Time to encode a single $512 \times 512 \times 16$ stack over bitrate.

X5550 clocked at 2.66GHz took 3.75 seconds. This exemplary point of the rate-distortion curve shows subjectively high image fidelity, as attested by our users, and is reasonably fast to compress. Since we process the data in disjoint stacks, the algorithm was trivially parallelized using OpenMP. We achieved a speedup of 7.2× on a workstation with two quad-core Xeon X5550. Clearly, the time required for encoding the data can be hidden in the time it takes to measure the data set.

7.2 User Study

We have conducted an informal user study to assess the efficacy and usability of the GPU-accelerated display-aware image viewer for digital pathology. Two experienced pathologists participated in the study. We set up three tasks: (1) a qualitative comparison of the general 2D browsing capability of our system and that of a widely used commercial slide-image viewer system [29], (2) a qualitative comparison of interactive focusing capabilities of the two viewer systems, and (3) a quantitative measurement of the speed with which a diagnosis can be made using the two viewer systems and using the traditional glass slide-based approach.

For task 1 and 2, both users agreed that our system outperforms the commercial viewer system. Specifically, interactivity and responsiveness are important factors affecting the pathologist's workflow. Our

system can zoom, pan, and change focus on large images almost instantaneously without noticeable lags that can affect the user's performance. A major complaint from both users was that the commercial system constantly showed significant delays to load images for view point changes. Importantly, the viewer software has an advantage over a light microscope because the slide image appears perfectly in focus at all times. When using a microscope, the pathologist must constantly change focus while moving a slide or changing the microscope objective because of issues such as objectives not being par-focal or the slide being slightly tilted. In this respect, the pathologists preferred using our system because it closely mimicked the performance of a microscope while also having the advantage of delivering a constantly sharp image.

For task 3, the pathologists were able to reach a diagnosis about 30–50% faster using our system than using the commercial system. However, they were able to render a diagnosis much faster using a light microscope than using computerized tools. We attribute this to the lack of training in the use of the viewer software and we expect that the speed of diagnosis will improve with experience. In fact, the users found it interesting to speculate how a highly efficient computer tool may impact their workflow once it is adopted and used regularly in clinical work.

The users provided positive feedback about the overall usability of our system: "much less time waiting for image to reload", "much more intuitive to use", "flexibility to zoom to exact magnification", "very close to driving slide by hand" were some of the comments we got. In their opinion, a mouse and keyboard interface is inappropriate for rapid inspection of slide images. With a more intuitive interface (e.g., a touch screen or a device that allows them to move their fingers as if they are moving a glass slide) they are convinced our system could be used for routine diagnostic work in their clinical setting.

To further demonstrate the flexibility and scalability of our display-aware framework, Figure 11 shows a screenshot of our display-aware framework rendering a 160 gigapixel electron micrographs.

7.3 Limitations

If the distance between object and viewer becomes too large, a given view might eventually comprise all stacks of the coarsest-available scale. Due to the vast amount of stacks and the large zoom range handled by our framework, these stacks will also become smaller than a single pixel on screen. Although one could argue that the users might not be interested in these zooms anyway, we think of addressing this issue in the future by merging coarse tiles on-the-fly. Assuming reasonable zoom speeds, our prefetching and caching system will then automatically alleviate the latencies arising during such an operation.

In this paper we did not address perceptual color spaces, since there is no perceptual space readily available for the grayscale electron microscope data. However, the results of Ljung et al. [23] clearly show the benefits of such metrics. Since these metrics do not impose any additional demands on the decoder because color re-conversion to RGB can be performed already in the encoding stage, we would like to explore such metrics in the future.

Users mildly criticized minor color differences that may arise between image stacks. These color differences are due to the fact that image stacks are non-overlapping during encoding. Therefore, neighboring stacks use disjoint codebooks that may fail in terms of perfect color reproduction. In the future, we will try to resolve this issue by providing the encoder with side-information on neighboring tiles (i.e., effectively provide an overlap between tiles that is removed after encoding).

8 Conclusions and Future Work

We have presented a framework for digital biomedical histology. In comparison to previous approaches, we remove the gap between acquisition and analysis using a display-aware approach that seeks to defer processing of image stacks until they are viewed by the user. To avoid frequent re-computation of this deferred processing, the results are cached and re-used during the next session. An important aspect in the context of data archival is that we never touch the original data.

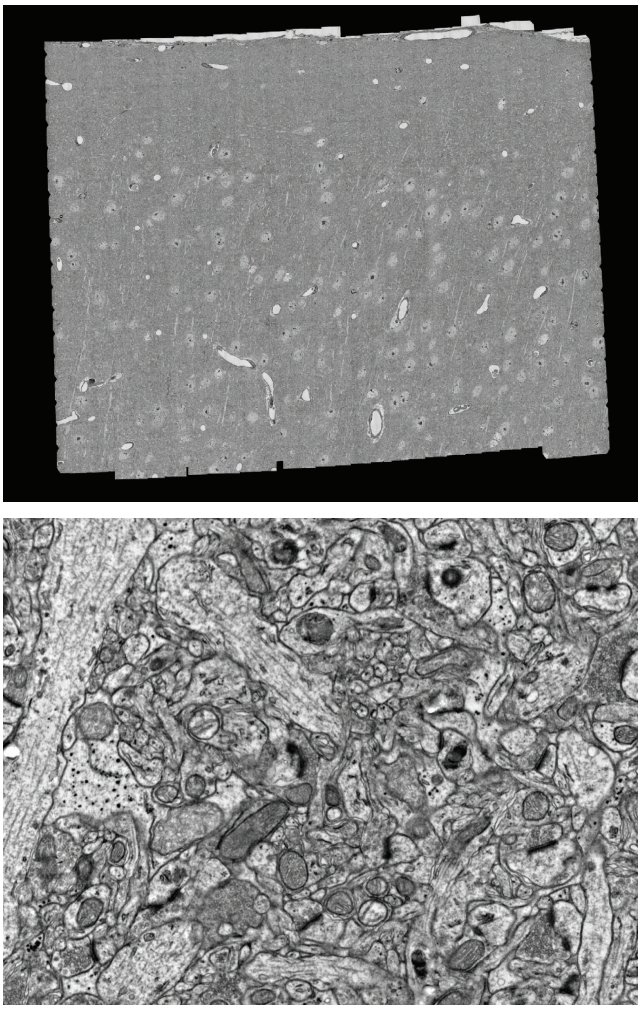

Fig. 11. A 160 gigapixel electron microscopy display stack rendered at interactive framerates using our display-aware framework. This biological image stack comprises 10 slices with 16 gigapixels each.

Instead, the originally measured data is only augmented with side information. This side information includes our compressed representation. Hence, we can always provide the user with an uncompressed view of her or his data.

We provide pathologists with a viewer for biomedical image stacks that for the first time combines the ability to interactively and rapidly zoom, pan, and change focus in the same manner as the traditional setup using a microscope. This is achieved by a novel variation on predictive hierarchical vector quantization that can be fully decoded on the GPU. Special attention was paid during the encoder design to keep the compressor fast enough for this specific setting.

Given the raw floating point performance of today's GPUs, it is an interesting future direction to run (semi-)automatic segmentation, classification, and filtering methods in a display-aware fashion—that is, directly on the GPU and only for the currently visible part of the data. This is particularly appealing as such operations can utilize the functionality of the presented framework without changes.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. PHY-0835713, the National Institutes of Health under Grant No. 1P30NS062685-01 and R01 NS020364-23, The Gatsby Charitable Foundation under Grant GAT3036-Connectomic Consortium, and through generous support from Microsoft Research and NVIDIA. We wish to thank Dr. Su-Jean Seo for participating in the user study.

REFERENCES

[1] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed textures. In *Proc. SIGGRAPH '96*, pages 373–378, 1996.

[2] D. Benson and J. Davis. Octree textures. In *Proc. SIGGRAPH '02*, pages 785–790, 2002.

[3] M. J. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, Mar. 1984.

[4] P. J. Burt and E. H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, 1983.

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

[6] N. Fout, K.-L. Ma, and J. Ahrens. Time-varying, multivariate volume data reduction. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1224–1230, 2005.

[7] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proc. SIGGRAPH '00*, pages 249–254, 2000.

[8] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Press, 1st edition, 1991.

[9] M. H. Ghavamnia and X. D. Yang. Direct rendering of laplacian pyramid compressed volume data. In *Proceedings of IEEE Visualization*, pages 192–199, 1995.

[10] GigaPan. `http://www.gigapan.org/`.

[11] J. Gilbertson and Y. Yagi. Histology, imaging and new diagnostic workflows in pathology. *Diagnostic Pathology*, 3(Suppl):S14, Jan 2008.

[12] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, 1998.

[13] S. Guthe and W. Straßer. Real-time decompression and visualization of animated volume data. In *IEEE Visualization*, pages 349–356, 2001.

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

[15] E. Haber and J. Modersitzki. Numerical methods for image registration. *Inverse Problems*, 24, 2004.

[16] J. Ho, A. Parwani, D. Jukic, Y. Yagi, and L. Anthony. Use of whole slide imaging in surgical pathology quality assurance: design and pilot validation studies. *Human pathology*, 37:322–331, Jan 2006.

[17] Aperio image scope. `http://www.aperio.com/pathology-services/imagescope-slide-viewing-software.asp`.

[18] J. Kopf, M. Uyttendaele, O. Deussen, and M. F. Cohen. Capturing and viewing gigapixel images. *ACM Transactions on Graphics*, 26(3):93:1–93:10, July 2007.

[19] M. Kraus and M. Strengert. Pyramid filters based on bilinear interpolation. In *Proceedings GRAPP 2007*, pages 21–28, 2007.

[20] Krieken, M. G. Rojo, I. Moreno, A. Ariza, and S. Tuzlali. A European network for virtual microscopy—design, implementation and evaluation of performance. *Virchows Archiv*, 454(4):421–429, Jan 2009.

[21] E. Krupinski, A. Tillack, L. Richter, and J. Henderson. Eye-movement study and human performance using telepathology virtual slides. implications for medical education and differences with experience. *Human pathology*, 37:1543–1556, Jan 2006.

[22] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28:84–94, 1980.

[23] P. Ljung, C. Lundström, A. Ynnerman, and K. Museth. Transfer function based adaptive decompression for volume rendering of large medical data sets. In *IEEE Symposium on Volume Visualization and Graphics*, pages 25–32, 2004.

[24] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, March 1982.

[25] E. Lum, K.-L. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. In *IEEE Visualization*, pages 263–270, 2001.

[26] K.-L. Ma and H.-W. Shen. Compression and accelerated rendering of time-varying volume data. In *Proceedings of the International Computer Symposium—Workshop on Computer Graphics and Virtual Reality*, pages 82–89, 2000.

[27] Microsoft HD view. `http://research.microsoft.com/en-us/um/redmond/groups/ivm/HDView/`.

[28] D. Nagayasu, F. Ino, and K. Hagihara. A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics*, 32(3):350–362, June 2008.

[29] NDP.view. `http://sales.hamamatsu.com/de/produkte/system-division/virtual-microscopy/products/software.php`.

[30] P. Ning and L. Hesselink. Vector quantization for volume rendering. In *Proceedings of the Workshop on Volume Visualization*, pages 69–74, 1992.

[31] Oberhumer.com. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`.

[32] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2nd edition, 2000.

[33] J. Schneider. Kompressions- und Darstellungsverfahren für hochaufgelöste Volumendaten. Diploma thesis, RWTH Aachen, Germany, 2003. http://wwwcg.in.tum.de/Research (English).

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

[35] Seadragon. `http://www.seadragon.com/`.

[36] H.-W. Shen and C. R. Johnson. Differential volume rendering: A fast volume visualization technique for flow animation. In *IEEE Visualization*, pages 180–187, 1994.

[37] C. Wang, H. Yu, and K.-L. Ma. Application-driven compression for visualizing large-scale time-varying data. *IEEE Computer Graphics and Applications*, 30(1):59–69, 2010.

[38] R. Westermann. Compression domain rendering of time-resolved volume data. In *IEEE Visualization 1995*, pages 168–175, 1995.

[39] J. Wilhelms and A. V. Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *In Proceedings of the 1994 Symposium on Volume Visualization*, pages 27–34, 1994.