Application-Driven Compression for Visualizing Large-Scale Time-Varying Data

Chaoli Wang ■ *Michigan Technological University*

Hongfeng Yu ■ *Sandia National Laboratories*

Kwan-Liu Ma ■ *University of California, Davis*

Leveraging the power of high-performance supercomputers and advanced numerical algorithms, scientists can perform 3D direct numerical simulations of many complex phenomena in unprecedented detail, leading to new scientific discoveries. Nowadays, a typical scientific simulation might produce data containing several hundred million voxels, hundreds of time steps, and tens of variables. The vast amounts of data generated from simulations present a challenge to data visualization. Even though the past several years witnessed great advancements in commodity graphics hardware, the video memory in graphics-hardware accelerators is still limited to around one gigabyte. Transferring data from a disk to the main memory and from the main memory to the video memory thus remains a key performance bottleneck for large-data visualization.

To address this issue, researchers have proposed different approaches to reducing the data sent through the graphics pipeline. These include different multiresolution data representations and a variety of data-reduction techniques, such as quantization and transform-based compression. Many algorithms, however, don't explicitly consider domain knowledge or the visualization tasks when representing and reducing data. Although these algorithms are general and easy to use for different data sets, they're not tightly coupled with specific visualization needs. So, the trade-off between reduction efficiency and visualization quality isn't ideal.

We advocate an application-driven approach to compressing and rendering large-scale time-varying scientific-simulation data. Scientists often have specific visualization tasks in mind based on certain domain knowledge. For example, in the context of time-varying, multivariate volume-data visualization, a scientist's domain knowledge might include the salient isosurface of interest for some variable. Given this knowledge, the scientist might want to observe spatiotemporal relationships among other variables in the neighborhood of that isosurface. We've tried to directly incorporate such knowledge and tasks into data reduction, compression, and rendering. Here, we present our solution and experimental results for two large-scale time-varying, multivariate scientific data sets.

An application-driven approach to compressing large-scale time-varying volume data achieves high compression rates and interactive rendering while preserving fine details surrounding regions of interest. Such an approach could help computational scientists cope with the large-data problem.

Algorithm Overview

Figure 1 illustrates the flowchart of our application-driven compression and rendering approach. (For information on related approaches, see the "Compression Methods for Time-Varying Volume Data" sidebar.)

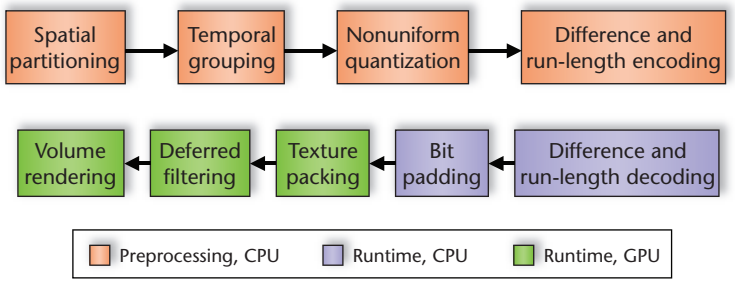

Figure 1. The flowchart of our application-driven compression and rendering approach. Our approach leverages a reference feature to partition the data for distance-based compression, which works hand-in-hand with traditional compression techniques for a greater amount of savings.

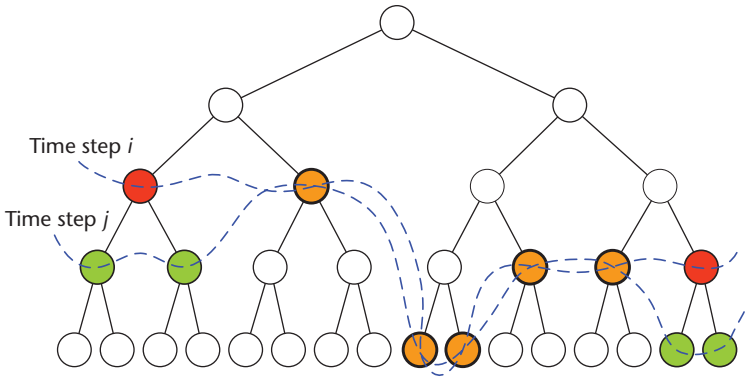

Figure 2. We partition the volume data at a time step into a list of different-sized blocks. The corresponding octree nodes constitute a cut through the full octree skeleton. Two neighboring time steps *i* and *j* share a subset of data blocks (drawn in orange), which we merge into space-time blocks in temporal groupings. For illustration purposes, we show a binary tree instead of an octree.

During preprocessing, we compress a given large time-varying data set. The compression starts with spatial partitioning and temporal grouping, which take into account domain knowledge that scientists provide. We then partition the data set into space-time blocks for individual compression. At runtime, we partially decode the compressed data blocks and perform bit padding in the CPU for texture loading. Then, we pack data blocks into the graphics memory and perform deferred filtering to reconstruct and render the data in the GPU. Our solution significantly reduces the amount of data transferred and efficiently uses the limited graphics memory.

Our Compression Scheme

Our scheme comprises the following three steps.

Spatial Partitioning

We assume that scientists know the regions of interest in their data. (For an example of the turbulent-combustion simulation data, see the related sidebar.) To measure each voxel's distance from the region of interest, we create a distance volume for each time step, where each voxel's distance value is the shortest distance between the voxel and the surface of interest.

We implement an algorithm similar to the fast marching method.[1] (The FMM is a technique for computing the arrival time of a front (that is, a moving contour) expanding in the normal direction at a set of grid points.) Instead of calculating the actual surface, we start with a front containing voxels that intersect with the surface and push the front outward gradually. We calculate the voxel-wise shortest distance for each voxel on the current front as an approximation of the distance to the actual surface. We can temporarily keep the distance volume in memory, or we can compress it and use it during runtime—for distance-based rendering, for example.

During spatial partitioning, we build an octree skeleton with a predefined block dimension for leaf nodes. Then, we start from the root node in the octree and partition the volume. If the data associated with the octree node doesn't include any isosurface voxels (voxels that intersect with the surface), or if the data block contains more than a certain percentage of isosurface voxels, then we don't partition the block any further. Otherwise, we partition the data block into eight subblocks and perform this process recursively until we arrive at the leaf nodes. In this way, we partition the entire volume at a time step into blocks of different sizes. The corresponding octree nodes constitute a cut through the full octree skeleton (see Figure 2).

After spatial partitioning, we calculate each data block's importance value, which is inversely proportional to the average distance values of all the voxels in that block. Our experiments show that regions closer to the surface of interest usually have finer partitioning. So, rather than using the conservative minimum distance value of all voxels in a data block to calculate the importance value, we use the average distance for a more aggressive compression afterward. In our implementation, we further scale the importance value using the ratio γ (<1.0), which decreases linearly as the average distance increases. Essentially, we use γ to steer the compression rate of space-time blocks.

Temporal Grouping

Spatial partitioning results in a list of different-sized blocks at each time step with different importance values. In general, the volume data at consecutive time steps exhibit strong temporal coherence in local neighborhoods. So, there's a large

Compression Methods for Time-Varying Volume Data

A wealth of research exists on volume data compression and rendering. Here, we review only related work on time-varying data visualization. Han-Wei Shen and Christopher Johnson have proposed differential volume rendering that uses temporal coherence between consecutive time steps to compress the data and accelerate volume animation.[1] Shen, along with Ling-Jen Chiang and Kwan-Liu Ma, has also described an approach dealing with large-scale time-varying fields.[2] The data structure, called the time-space partitioning (TSP) tree, captures the data's spatial and temporal coherence. The TSP tree uses an octree for spatial partitioning and a binary tree for storing temporal information at each octree node. Both approaches treat the spatial and temporal dimension separately. Alternatively, we can treat temporal and spatial dimensions uniformly. For example, Jane Wilhelms and Allen Van Gelder encoded time-varying data using a multidimensional (4D) tree.[3]

Other research efforts in time-varying data visualization have focused on transform-based compression and rendering. Stefan Guthe and Wolfgang Straßer introduced an algorithm that uses the wavelet transform to encode each spatial volume, then applies a motion compensation strategy to match the volume blocks in adjacent time steps.[4] Eric Lum and his colleagues proposed using the discrete cosine transform to encode individual voxels along the time dimension.[5] They employed a color table animation technique to render the volumes using texture hardware. Bong-Soo Sohn and his colleagues described a compression scheme that uses the wavelet transform to create intracoded volumes and applies difference encoding to compress the time sequence.[6] Jens Schneider and Rudiger Westermann presented a hierarchical vector quantization solution to compress time-varying volumetric data, performing both decompression and rendering at runtime in graphics hardware.[7] Nathaniel Fout and his colleagues also used vector quantization for time-varying, multivariate volume-data reduction that exploits correlations among related variables.[8]

Research work closely related to ours includes that of Ma and his other colleagues[9] and of Chandrajit Bajaj and his colleagues.[10] Ma and his colleagues used octree encoding and difference encoding for spatial and temporal domain compression, respectively. They also investigated how quantization might affect further compression, rendering optimization, and image results. Bajaj and his colleagues classified voxels according to their importance in visualization, and assigned weights to them. To compress the volume data, they used the Haar wavelet transform and defined weight functions for wavelet coefficients based on voxels' weights in the same spatial-frequency locations.

References

1. H.-W. Shen and C.R. Johnson, "Differential Volume Rendering: A Fast Volume Visualization Technique for Flow Animation," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 1994, pp. 180–187.
2. H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 1999, pp. 371–377.
3. J. Wilhelms and A. Van Gelder, "Multi-dimensional Trees for Controlled Volume Rendering and Compression," *Proc. IEEE Symp. Volume Visualization*, IEEE CS Press, 1994, pp. 27–34.
4. S. Guthe and W. Straßer, "Real-Time Decompression and Visualization of Animated Volume Data," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 2001, pp. 349–356.
5. E.B. Lum, K.-L. Ma, and J. Clyne, "Texture Hardware Assisted Rendering of Time-Varying Volume Data," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 2001, pp. 263–270.
6. B.S. Sohn, C.L. Bajaj, and V. Siddavanahalli, "Feature Based Volumetric Video Compression for Interactive Playback," *Proc. IEEE Symp. Volume Visualization*, IEEE CS Press, 2002, pp. 89–96.
7. J. Schneider and R. Westermann, "Compression Domain Volume Rendering," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 2003, pp. 293–300.
8. N. Fout, K.-L. Ma, and J.P. Ahrens, "Time-Varying Multivariate Volume Data Reduction," *Proc. ACM Symp. Applied Computing*, ACM Press, 2005, pp. 1224–1230.
9. K.-L. Ma et al., *Efficient Encoding and Rendering of Time-Varying Volume Data,* ICASE report 98-22, Inst. Computer Applications in Science and Eng., 1998.
10. C.L. Bajaj, S. Park, and I. Ihm, "Visualization-Specific Compression of Large Volume Data," *Proc. Pacific Graphics*, IEEE CS Press, 2001, pp. 212–222.

degree of node overlap in the block lists for neighboring time steps, as Figure 2 shows.

To use this temporal coherence for compression, for each octree node, we merge spatial blocks at consecutive time steps into space-time blocks. Meanwhile, we specify a maximum window size w to control the trade-off between compression rate and decompression speed. Figure 3 shows an example of a temporal grouping on an octree node. In essence, temporal grouping consolidates data blocks at different time steps into space-time blocks, which become the basic units for the following encoding.

Encoding

We compress all the space-time blocks using nonuniform quantization together with difference and run-length encoding, resulting in a highly compacted data representation. We first create a

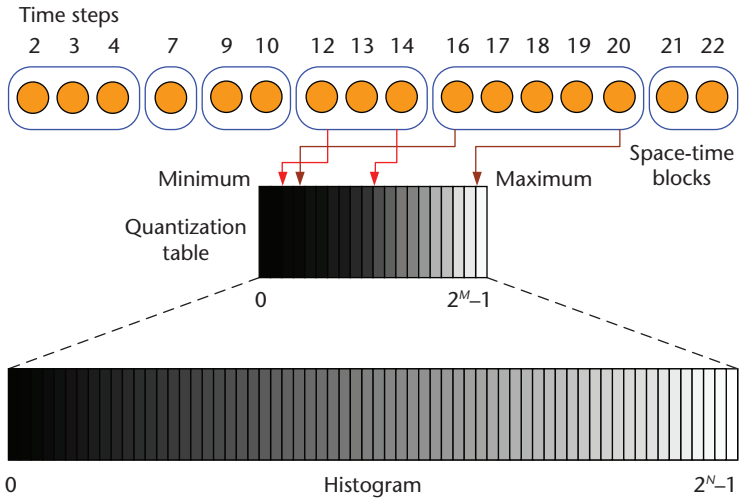

Figure 3. Temporal grouping merges spatial blocks in the same octree node at consecutive time steps into space-time blocks. In this example, the maximum window size is five. For each space-time block, we record the beginning and ending indices with respect to the nonuniform quantization table and the offset index for each voxel in the block. In the figure, a time step (such as 5) that doesn't appear in the octree node stays on one of its ancestor or descendant nodes instead.

histogram (with 2^N entries) from the volume data at all time steps. Then, we build a nonuniform quantization table (with 2^M entries, where $M < N$) from the histogram.

We can apply many solutions for histogram quantization. One simple yet effective solution is to partition the histogram into 2^M parts with equal areas (that is, accumulated bin counts) and pick the data value with the highest bin count (the most frequently occurring value) in each part as the quantized value.[2] Furthermore, our quantization also incorporates data-specific domain knowledge, such as ranges of interest. This ensures that particular data ranges of interest are sufficiently sampled and represented in the quantization table. The encoding works as follows. For each space-time block, we use the minimum and maximum values to retrieve the beginning and ending indices with respect to the quantization table, as Figure 3 shows. For each voxel in the space-time block, recording the offset index (I_o) with respect to the beginning index (I_b) is sufficient to look up the corresponding quantized value. Moreover, from the beginning and ending indices, we can calculate the number of bits (B_n) needed to encode each voxel's offset index.

The number of bits is further modulated by the average distance of the space-time block to the surface of interest. That is, the farther a block is from the surface, the fewer bits we use to represent each voxel in the block. Adjusting relevant parameters such as γ gives different bit modulations for space-time blocks and thus different compression levels.

Let B_m represent the number of bits after distance modulation, where $B_m \leq B_n$. The actual index I with respect to the quantization table is

$$I = I_b + I_o \times 2^{Bn \sqsubset Bm}. \quad (1)$$

After the quantization step, each voxel in the same space-time block is represented with a few bits. We can further compress the entire space-time block using a combination of difference and run-length encoding. First, we calculate the differences of index values for neighboring time steps in each space-time block. Then, we can compress the difference values using run-length encoding to exploit temporal coherence.

Our Decompression and Rendering Scheme

At runtime, we partially decompress the compressed time-varying data using difference and run-length decoding. As Figure 1 shows, we conduct this step in the CPU to produce space-time blocks storing offset indices with respect to the quantization table. To render volume data at a time step, the corresponding data blocks are categorized, padded, and packed into texture memory. Finally, we render the volume using deferred filtering.

Bit Padding

Assume that the nonuniform quantization table has up to 1,024 entries. So, the offset indices stored in a space-time block could range from 1 to 10 bits. The number of bits representing offset indices varies depending on the block's data range and importance value. However, standard OpenGL defines only fixed formats for the pixel data for texture loading, so we picked the formats listed in Table 1. Clearly, this bit-padding step involves space overhead and increases texture-memory consumption, but our experiments show that the size increase is affordable at approximately 10 percent.

Texture Packing

After bit padding, we're ready to pack the data blocks into one of the five index textures (one for each of the internal texture formats listed in Table 1). Texture packing reduces the number of textures used and lowers texture-memory consumption (for more information, see the "Texture Packing" sidebar). We perform texture packing by treating each data block as a 3D array and by using a greedy algorithm to optimize block arrangement.

When we use deferred filtering during decompression, we're reconstructing volume slices voxel by voxel using the nearest-neighbor interpolation

rather than the trilinear interpolation. So, in our scenario, we don't have to perform texture packing directly in 3D. Instead, we can treat each 3D data block as a 1D array and pack it into a 3D texture in the form of a cube. We calculate the packed 3D texture's dimension as

$$L = \lceil \sqrt[3]{\|A\|} \rceil,$$

where $||A||$ denotes the length of array A. This treatment greatly simplifies data packing and better uses the texture memory. The only overhead is that we need to map (x, y, z) tuples to 1D indices for texture lookup.

Deferred Filtering and Volume Rendering

At runtime, we decompress each voxel in the volume for rendering, performing the trilinear interpolation on a per-fragment basis. A single voxel might be needed multiple times for neighboring sample reconstructions and gradient calculations. To avoid redundant decompressions, we can first cache a proxy geometry (a small subset of decompressed volume), then use the subset for conventional rendering. This deferred-filtering technique separates decompression and interpolation into two passes so that we need to decompress a voxel only once no matter how many times it's needed for interpolation.[3]

In our case, the proxy geometry is multiple axis-aligned slices assembled from volume partitions (see Figure 4). To render a slab, we decompress two consecutive slices of the volume in the first pass. In the second pass, we render sampling slices using trilinearly interpolated samples. We thus render the volume slab by slab. The main advantage of using axis-aligned slices is that we don't need to pad data blocks in the volume to ensure seamless rendering along block boundaries.

Figure 4 sketches our texture lookups using deferred-filtering. We dynamically reconstruct the axis-aligned slices most perpendicular to the viewing direction. The address texture stores each data block's address in the packed index texture.

Turbulent-Combustion Simulation

Sandia National Laboratories scientists have performed terascale turbulent-combustion simulation to study the basic phenomena of reacting flows in the combustion process.[1] Of scientific interest is the main flame structure, which corresponds to the stoichiometric mixture fraction (*mixfrac*) surface at an isovalue of 0.2 (see Figure A). In particular, scientists hope to observe how other variables distribute along the main flame surface—critical knowledge for evaluating the combustion process's efficiency.

This kind of analytical visualization is quite common in scientific data analysis. Such domain knowledge should be translated into a reference feature to guide our application-driven data compression and rendering. The intuition is that the closer a voxel is to the surface of interest, the higher precision we should preserve to ensure reconstruction quality. In other words, the precisions of data should vary according to their associations to the reference feature.

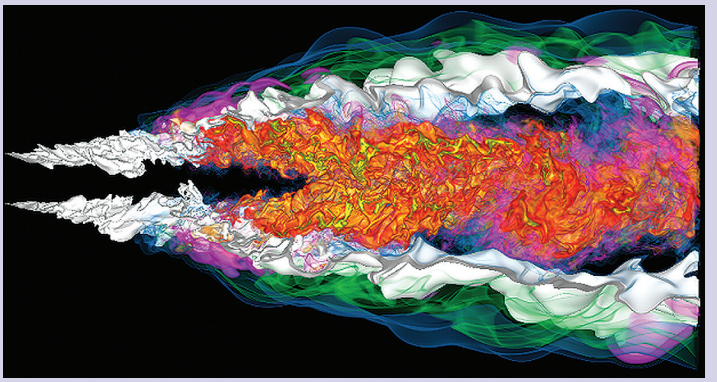

Figure A. Simultaneous rendering of two variables of the turbulent-combustion simulation data set. The mixture fraction (*mixfrac*) surface (at an isovalue of 0.2) is white; the HO_2 variable is depicted in a volume-rendering style. Our work exploits such a surface of interest to compress the data effectively.

Reference

1. E.R. Hawkes et al., "Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights towards Predictive Models," *J. Physics: Conf. Series*, vol. 16, 2005, pp. 65–79.

For each data block on a sampling slice, we first look up its block address. We also look up the beginning index in the address texture and the bit

Table 1. A space-time block is padded into one of the five different OpenGL pixel data formats for texture loading.

Original no. of bits	No. of padded bits	No. of elements in a texel	Overhead (%)	Data type for pixel data	Internal texture format
1, 2	2	3	166.7, 33.3	GL_UNSIGNED_BYTE_3_3_2	GL_R3_G3_B2
3, 4	4	4	33.3, 0.0	GL_UNSIGNED_SHORT_4_4_4_4	GL_RGBA4
5	5	3	6.67	GL_UNSIGNED_SHORT_5_5_5_1	GL_RGB5_A1
6, 7, 8	8	4	33.3, 14.3, 0.0	GL_UNSIGNED_INT_8_8_8_8	GL_RGBA8
9, 10	10	3	18.5, 6.67	GL_UNSIGNED_INT_10_10_10_2	GL_RGB10_A2

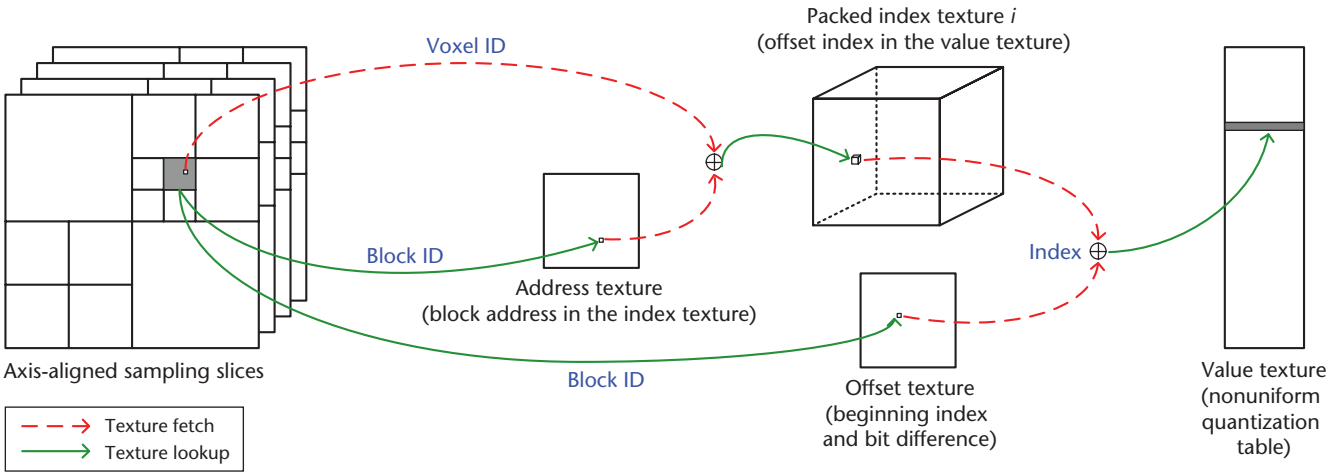

Figure 4. Texture lookups with the deferred-filtering scheme. The address texture stores data blocks' addresses in the packed index texture. The offset texture stores data blocks' beginning indices and bit differences. The packed index texture stores voxels' offset indices with respect to the value texture (that is, the nonuniform quantization table). To reconstruct a data block on a sampling slice, we need two texture lookups on a per-block basis and two on a per-voxel basis.

Table 2. The two data sets and their experimental results.

	Combustion data set	Hurricane data set
Volume dimension	(800, 686, 215)	(500, 500, 100)
No. of time steps	53	48
No. of variables	4	4
Data size	92.3 Gbytes	17.9 Gbytes
Block dimension	(64, 64, 32)	(32, 32, 16)
Average no. of nodes	278	275
Average node overlap	90%	62%
Compressed size (after quantization)	15.12 Gbytes	3.6 Gbytes
Compressed size (after difference and run-length encoding)	4.53 Gbytes	900 Mbytes
Data reduction on disk	20.57×	20.37×
Compression time	3 hrs.	40 min.
Padding overhead	11%	12%
Texture reduction on GPU	82%	77%
Frame rate*	12.5 fps	28.5 fps

** We measured the frame rate (in frames per second) for rendering one variable with a 512^2 viewport and a sampling rate of 1.0.*

difference ($B_d = B_n - B_m$) in the offset texture. Then, for each voxel in the data block, we look up its offset index in the corresponding packed index texture using its voxel ID and the block address. Finally, we use the voxel's offset index and the block's beginning index and bit difference to compute the actual index to the value texture (Equation 1).

Results

We experimented with our algorithm on two floating-point data sets obtained from scientific simulations (see Table 2). We performed all tests on a PC with a 2.33-GHz Intel Xeon processor with 4 Gbytes of main memory and an Nvidia GeForce 8800 GTX graphics card with 768 Mbytes of video memory.

Sandia National Laboratories scientists provided the combustion data set. The combustion simulation ran thousands of time steps; at each time step it output dozens of variables representing different chemical species. A subset of the data set we used here has four variables: scalar dissipation rate (*chi*), stoichiometric mixture fraction (*mixfrac*), hydroperoxy radical (HO_2), and hydroxyl radical (OH). The scientific interest for the combustion data is on the main flame structure, which corresponds to the *mixfrac* surface, for which the isovalue is 0.2.

We obtained the hurricane data set from the US National Center for Atmospheric Research. The hurricane modeled in the simulation is Hurricane Isabel, a strong hurricane that occurred during September 2003 in the West Atlantic. For our ex-

periment, we picked four variables: pressure (*P*), cloud moisture (*Cloud*), total precipitation (*Precip*), and water vapor (*Qvapor*). We focused on the region with very low pressure ($P \approx 0$), which corresponded to the hurricane's center.

Compression

To construct the octree skeleton, we set the block dimension for leaf nodes as (64, 64, 32) for the combustion data set and (32, 32, 16) for the hurricane data set. For spatial partitioning, we chose five percent as the threshold for the percentage of isosurface voxels in the data blocks. We determined the block size for leaf nodes and the percentage threshold for isosurface voxels on the basis of the number of blocks generated during spatial partitioning.

With such configurations, the average number of octree nodes with nonempty data blocks in a time step was approximately 270. On the other hand, the average node overlap for consecutive time steps was 90 percent for the combustion data set and 62 percent for the hurricane data set. This indicates a great degree of coherence for compression. We set the maximum window size as five ($w = 5$) in temporal grouping. For both data sets, we chose $N = 16$ and $M = 10$ for the nonuniform quantization, which let us sufficiently sample the histogram in the 1,024-entry quantization table.

It took us three hours to compress the 93.2-Gbyte combustion data set (on average, less than one minute per variable per time step, which is approximately 450 Mbytes). The compressed data size was 4.53 Gbytes, so the compression rate was 20.57×. This means that, on average, we compressed each variable in the time sequence to approximately 1.6 bits per voxel. We achieved a comparable compression performance for the hurricane data set.

Figure 5 shows the signal-to-noise ratio (SNR) and the peak signal-to-noise ratio (PSNR) of the compressed hurricane data set. Different curves correspond to different distance ranges from the surface of interest for the *Cloud* (Figure 5a), *Precip* (Figure 5b), and *Qvapor* (Figure 5c) variables. Generally, the regions with smaller distances (those closer to the surface) got higher SNRs or PSNRs (less distortion). At some time steps (such as time steps 1 to 8 for the SNR curves and time steps 40 to 48 for the PSNR curves), this observation didn't always hold. This is because we employ blockwise compression instead of voxelwise compression and because we use the average distance, instead of the minimum distance, to calculate a block's importance value.

With these settings, the block size also mattered, because all voxels in a block used the same encoding scheme. On the other hand, Figure 5c shows that *Qvapor* had higher SNRs and lower PSNRs than *Cloud* and *Precip*. This suggests that different variables might require customized bit modulations for compressing their space-time blocks to balance the overall rate distortion across all the variables.

Decompression and Rendering

At runtime, we padded and loaded the partially decoded data into texture memory. As Table 2 indicates, bit padding only slightly increased memory usage. The overall texture reduction with respect to loading the original data was 82 and 77 percent for the combustion and hurricane data sets,

Texture Packing

To effectively use limited graphics memory, Martin Kraus and Thomas Ertl introduced adaptive texture maps with locally adaptive resolution.[1] They used these maps to pack data blocks of different resolutions. This technique lets us represent fine details in images and volumes without increasing the whole texture map's resolution.

Alécio Binotto and his colleagues developed a similar approach for texture packing and compression of sparse time-varying volume data into 3D textures.[2] During rendering, the fragment shader decompresses data in the GPU.

Wei Li and his colleagues studied texture partitioning and packing for skipping empty space and accelerating slice-based volume rendering.[3] They first partition the entire volume into subvolumes with similar properties. Then they pack and stitch together the subvolumes to create larger textures for rendering.

Hiroshi Akiba and his colleagues used data packing for time-varying data reduction.[4] To achieve data packing, they discarded data blocks with values outside the data interval of interest and encoded the remaining data such that they could efficiently decode it in the GPU.

References

1. M. Kraus and T. Ertl, "Adaptive Texture Maps," *Proc. ACM Siggraph/Eurographics Conf. Graphics Hardware*, Eurographics Assoc., 2002, pp. 7–15.
2. A.P.D. Binotto, J.L.D. Comba, and C.M.D. Freitas, "Real-Time Volume Rendering of Time-Varying Data Using a Fragment-Shader Compression Approach," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics*, IEEE CS Press, 2003, pp. 69–75.
3. W. Li, K. Mueller, and A.E. Kaufman, "Empty Space Skipping and Occlusion Clipping for Texture-Based Volume Rendering," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 2003, pp. 317–324.
4. H. Akiba, K.-L. Ma, and J. Clyne, "End-to-End Data Reduction and Hardware Accelerated Rendering Techniques for Visualizing Time-Varying Non-uniform Grid Volume Data," *Proc. 4th Int'l Workshop Volume Graphics*, IEEE CS Press, 2005, pp. 31–39.

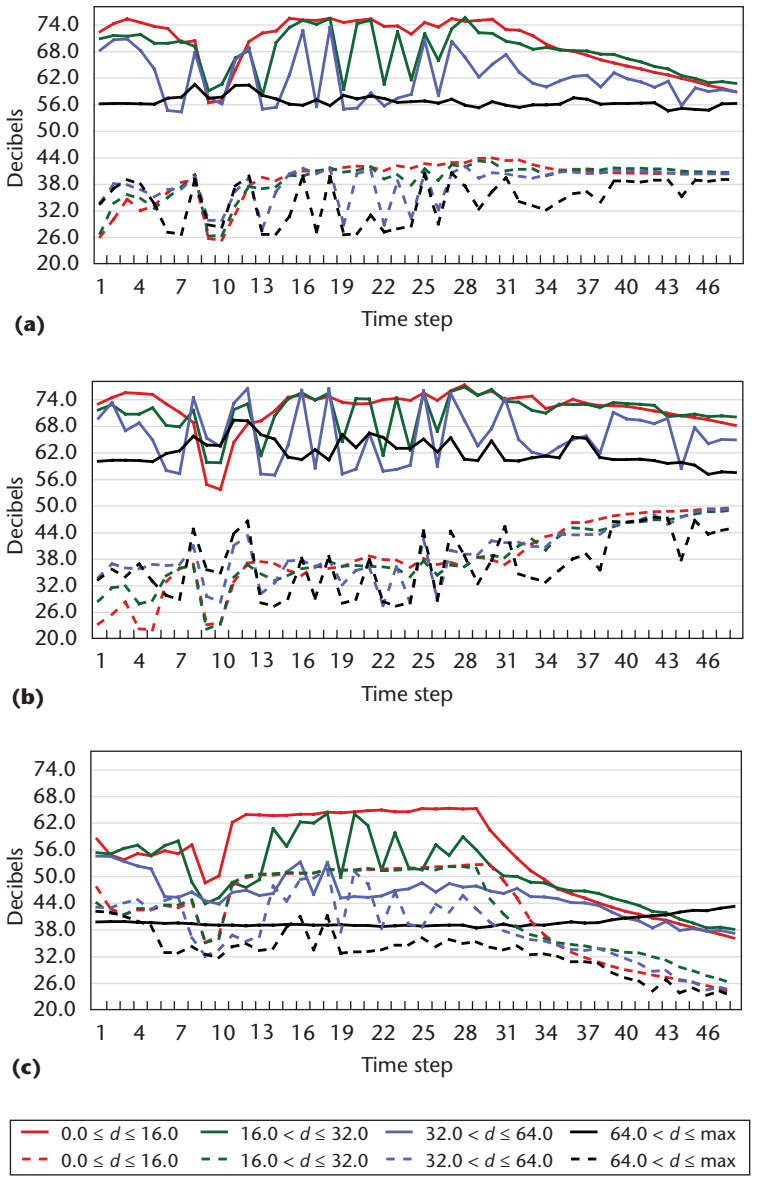

Figure 5. The signal-to-noise ratio (SNR) and the peak signal-to-noise ratio (PSNR) curves of the application-driven compressed hurricane data set. We show four SNR (dashed lines) and PSNR (solid lines) curves with different distance ranges from the surface of interest for the (a) *Cloud,* (b) *Precip,* and (c) *Qvapor* variables. We used the theoretical signal peak as a reference in PSNR calculation. In general, the regions close to the surface yield higher SNRs or PSNRs and thus less distortion.

respectively. Owing to difference and run-length decoding and bit padding, we reduced the compression rates in the texture memory to 5.6× and 4.3× for the combustion and hurricane data sets, respectively.

With a 512^2 viewport and a regular sampling rate of 1.0 (one sample per voxel), we achieved 12.5 frames per second (fps) for rendering one variable from the combustion data set and 28.5 fps for rendering one variable from the hurricane data set (including deferred filtering and volume rendering). This performance is comparable to conventional volume rendering. Our application-driven compression and rendering solution makes interactive visualization of large-scale time-varying data possible, while dramatically reducing data transferring between the memory hierarchies.

Table 3. Timing breakdown for rendering the *Cloud* variable of the 48 time-step hurricane data set.

Compression and rendering stage	Time (sec.)
Difference and run-length decoding	2.37
Bit padding	2.58
Texture packing	1.27
Deferred filtering and volume rendering	2.95

The savings in data transferring greatly shorten the time to animate time-varying data. For example, we reduced the total time (including I/O and rendering) for animating the *Cloud* variable of the 48 time-step hurricane data set from 36.96 seconds to 9.17 seconds. Table 3 breaks down the timing for each stage: decoding, bit padding, texture packing, deferred filtering, and volume rendering. (The decoding stage includes the time to read compressed data from the disk.) The frame rate improved from 1.3 fps (without compression) to 5.2 fps (with compression), achieving a highly desirable level of interactivity.

Figure 6 compares the rendering of the compressed data (Figures 6a and 6b) to the original data (Figure 6c). To render more than one variable simultaneously over time, we independently decompressed each variable and loaded it into the graphics card. For objective comparison, we calculated pixelwise differences (the Euclidean distances) of images generated from the compressed and original data in the CIELUV color space (see Figure 6d). We mapped the noticeable pixel differences (with $\Delta E \geq 4.0$) to nonwhite colors in Figure 6e (clamping differences greater than 255).

Our application-driven solution preserves fine details near the regions of interest, while maintaining the overall quality. We can perceive some visual differences between the rendering of the compressed versus original data. However, these correspond to regions far from the reference feature, so they lost more precision during quantization.

Figure 7 shows the rendering of the compressed hurricane data set at different compression levels. Adjusting the parameters for bit modulation compresses the data with different reduction rates. Figure 7 shows how the quality degrades with the increased compression rate. Compared with the *Qvapor* variable at the same compression level, the *Cloud* variable gives less degradation in visual quality. We rendered the *P* surface with *Cloud* and *Qvapor*. The rendering let the scientists focus on

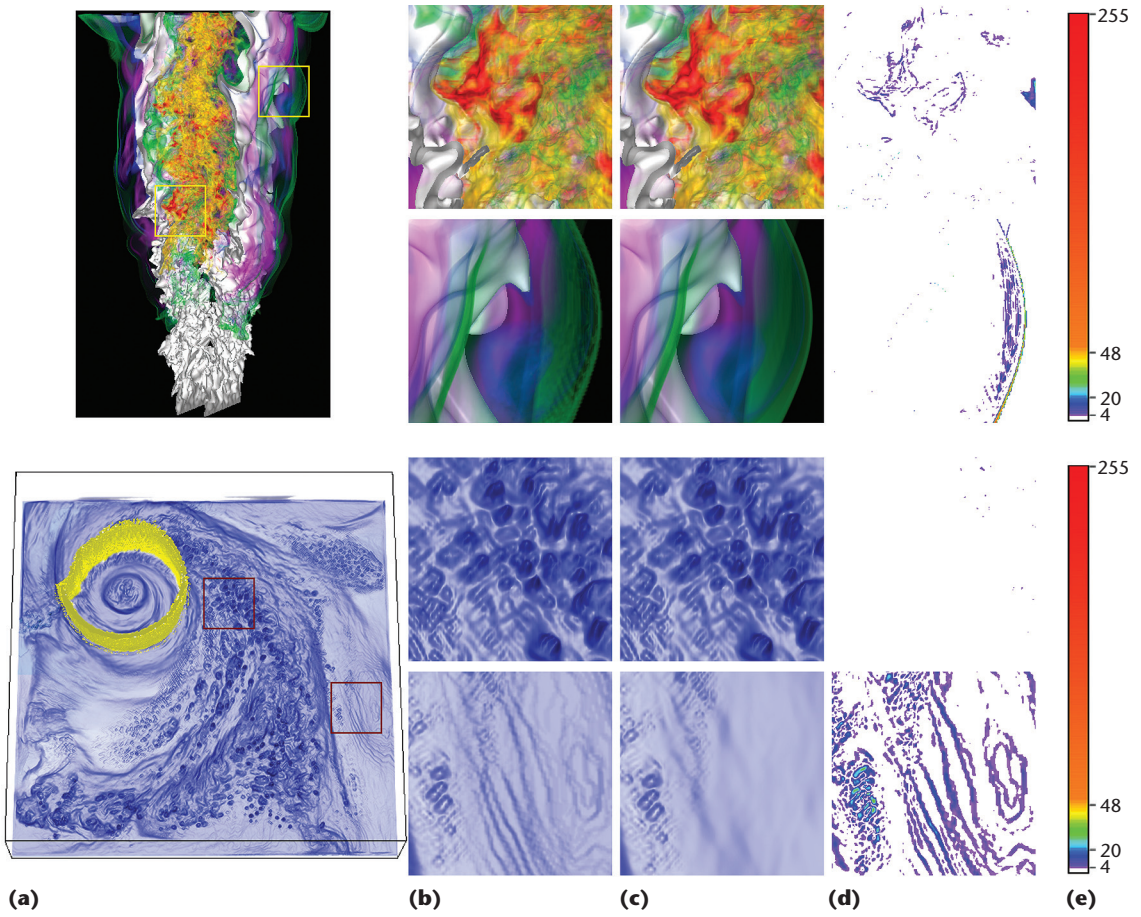

Figure 6. Comparing the compressed versus original data (the combustion data is on top; the hurricane data is on the bottom). We rendered (a) compressed data for an overview of the *mixfrac* surface (in white) plus the HO_2 variable (top image) and the *P* surface (in yellow) plus the *Qvapor* variable (bottom image). We also zoomed in on (b) the compressed data and (c) the original data. Finally, we show (d) the image difference of the compressed and original data calculated in the CIELUV color space and provide (e) a color map. Regions farther away from the surface of interest show more quantization artifacts in rendering.

the hurricane center and track other flow properties around the surface of interest.

For the combustion data set, where the variable rendering could occlude the surface, we could use the distance volume to perform flexible rendering by changing the distance threshold (see Figure 8). We assigned nonzero opacity values to only the voxels in the given distance threshold. A scientist can interactively control the amount of information displayed around the surface to better observe variable relationships at runtime. The accompanying videos (see the Web Extras section of www.computer.org/portal/web/computingnow/cga) show side-by-side rendering of the original and compressed combustion and hurricane data sets over all time steps.

Discussion

We opted for scalar quantization instead of vector quantization for data reduction because in vector quantization, the time to generate the codebook could be prohibitively long for a large time-varying, multivariate data set. The nonuniform quantization we implemented is simple and fast, and produced good compression and reconstruction results for the two test data sets.

With this quantization, however, we could miss details for underrepresented scalar values. This happens when the transfer function maps underrepresented scalar values to high opacity values. We could also use other quantization approaches, such as Lloyd's quantizer, which guarantees to converge to a local minimum in the L_2 metric. We need more research on quantization that couples compression with visualization to strive for a better trade-off between reduction performance and rendering quality.

Our approach resembles the importance-driven volume-rendering work by Ivan Viola and his colleagues.[4] However, we use the importance values of data blocks in relation to the surface of interest in compression and rendering. The limitation

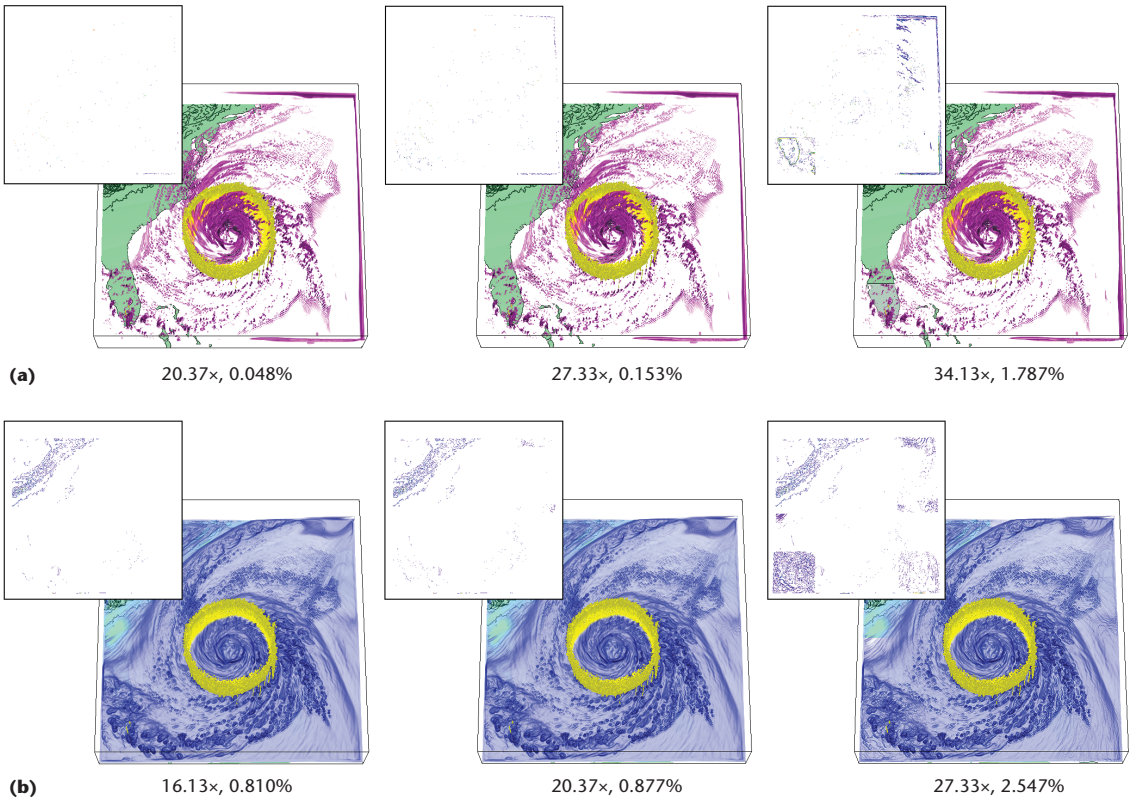

Figure 7. Rendering the compressed hurricane data set at time step 24 with different compression levels. By adjusting the parameters for bit modulation, we can change the number of bits allocated to each space-time block and compress the data differently. The larger boxes show the near-zero pressure surfaces with the (a) *Cloud* and (b) *Qvapor* variables. The smaller boxes show the corresponding difference image with respect to the original image. We provide the compression rate and the percentage of noticeable pixel difference.

of our approach is that because the reference feature derived from domain knowledge is explicitly incorporated into data reduction, we must redo the compression process if the scientific interest changes. We assume that such a change doesn't happen frequently. We could encode all possible input from scientists, letting users shift features of interest at runtime, but this would reduce compression efficiency.

Our application-driven approach clearly suggests a viable direction for addressing the data challenge presented by large-scale scientific simulations. We can apply our solution to other domains where the reference features are in different forms, such as vortices in the flow data. As long as the identified regions of interest occupy only a small percentage of the volume space, our method would remain effective.

We hope to incorporate our data-reduction scheme with multiresolution techniques to support flexible level-of-detail rendering. We'll also consider a parallel implementation of our solution for compressing and rendering large-scale time-varying data in a massively parallel computing environment.

Acknowledgments

This research was supported in part by the US National Science Foundation's Information Technology Research program and the US Department of Energy's Scientific Discovery through Advanced Computing program. We thank Jacqueline H. Chen at Sandia National Laboratories for providing the combustion data set and sharing her domain knowledge. We also thank the reviewers for their constructive suggestions.

References

1. M.W. Jones, J.A. Bærentzen, and M. Srámek, "3D Distance Fields: A Survey of Techniques and Applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 12, no. 4, 2006, pp. 581–599.
2. A.E. Shortt, T.J. Naughton, and B. Javidi, "Histogram Approaches for Lossy Compression of Digital Holograms of Three-Dimensional Objects," *IEEE Trans. Image Processing*, vol. 16, no. 6, 2007, pp. 1548–1556.
3. N. Fout et al., "High-Quality Rendering of Compressed Volume Data Formats," *Proc. Eurographics/IEEE VGTC Symp. Visualization*, Eurographics Assoc., 2005, pp. 77–84.
4. I. Viola, A. Kanitsar, and M.E. Gröller, "Importance-

Driven Volume Rendering," *Proc. IEEE Visualization Conf.*, IEEE CS Press, 2004, pp. 139–145.

Chaoli Wang *is an assistant professor of computer science at Michigan Technological University. He previously was a postdoctoral researcher at the University of California, Davis. His research focuses on large-scale data analysis and visualization, high-performance computing, and user interfaces and interaction. Wang has a PhD in computer and information science from Ohio State University. He's a member of the IEEE. Contact him at chaoliw@mtu.edu.*

Hongfeng Yu *is a postdoctoral researcher at Sandia National Laboratories. His research interests include scientific visualization and parallel computing. Yu has a PhD in computer science from the University of California, Davis. Contact him at hyu@sandia.gov.*

Kwan-Liu Ma *is a computer science professor at the University of California, Davis, and directs the US Department of Energy's SciDAC (Scientific Discovery through Advanced Computing) Institute for Ultrascale Visualization. His research spans visualization, high-performance computing, and user interface design. Ma has a PhD in computer science from the University of Utah. He serves on the editorial boards of* IEEE Computer Graphics and Applications *and* IEEE Transactions on Visualization and Computer Graphics. *He's a senior member of the IEEE and a member of the ACM. Contact him at ma@cs.ucdavis.edu.*

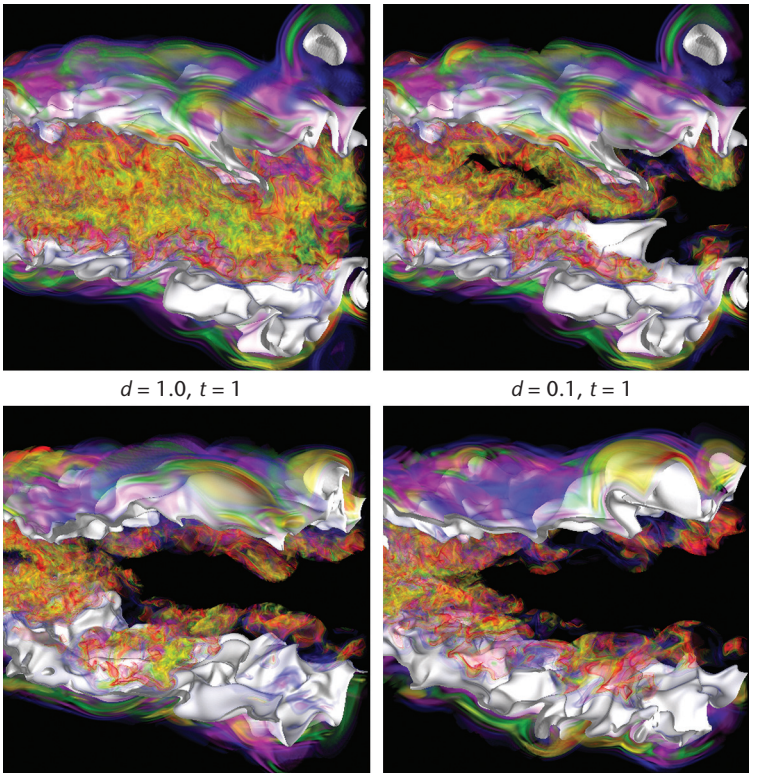

Figure 8. Rendering the compressed combustion data (the *mixfrac* surface with the HO_2 variable) with distance control. By changing the distance threshold (normalized to [0, 1]), the scientist can interactively control the amount of information displayed around the surface to better observe variable relationships.