A Decompression Pipeline for Accelerating Out-of-Core Volume Rendering of Time-Varying Data

Daisuke NAGAYASU [a,b] Fumihiko INO [a,*]
Kenichi HAGIHARA [a]

[a] *Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan*

[b] *Oracle Corporation Japan, 4-1 Kioi, Chiyoda, Tokyo 102-0094, Japan*

Abstract

This paper presents a decompression pipeline capable of accelerating out-of-core volume rendering of time-varying scalar data. Our pipeline is based on a two-stage compression method that cooperatively uses the CPU and GPU (graphics processing unit) to transfer compressed data entirely from the storage device to the video memory. This method combines two different compression algorithms, namely packed volume texture compression (PVTC) and Lempel-Ziv-Oberhumer (LZO) compression, allowing us to exploit both temporal and spatial coherence in time-varying data. Furthermore, it achieves fast decompression by taking architectural advantages of each processing unit: a hardware component on the GPU and a large cache on the CPU, each suited to decompress PVTC and LZO encoded data, respectively. We also integrate the method with a thread-based pipeline mechanism to increase the data throughput by overlapping data loading, data decompression, and rendering stages. Our pipelined renderer runs on a quad-core PC and achieves a video rate of 41 frames per second (fps) in average for $258 \times 258 \times 208$ voxel data with 150 time steps. It also demonstrates an almost interactive rate of 8 fps for $512 \times 512 \times 295$ voxel data with 411 time steps.

Key words: Volume rendering, time-varying data, pipelined rendering, data compression, GPU

* Corresponding author. Tel.: +81 6 6850 6597; fax: +81 6 6850 6599.
Email address: ino@ist.osaka-u.ac.jp (Fumihiko INO).

1 Introduction

Volume rendering [1] of time-varying data is capable of producing animation sequences that show how the three-dimensional (3-D) structure evolves over time. This visualization technique plays an increasingly important role for the intuitive understanding of complex time-varying phenomena [2]. For example, it is useful to interpret simulation results and scanned X-ray images in physical and life sciences. Thus, real-time rendering of time-varying volume data is required to assist scientists effectively in time-series analysis.

One challenging issue in time-varying volume visualization is to develop an efficient mechanism for handling 4-D data. For example, a time-varying volume of $512 \times 512 \times 512$ voxels with a hundred time steps requires 12.5 GB of memory if each voxel has 1-byte data. Moreover, recent advances in integrated circuits allow us to produce large-scale datasets, which could not be entirely stored in the main memory as well as in the video memory. Due to this increasing data size, we need fast out-of-core rendering systems, which relax memory requirements by on-demand loading of data.

Data compression techniques also provide an effective solution to the issue mentioned above. Actually, such techniques are integrated into many rendering systems [3–10] to reduce data size. Most of them use the graphics processing unit (GPU) [11] for data decompression. These GPU-based methods [3–7] are more efficient compared with CPU-based methods [8,9], which might suffer from low performance due to the raw data transferred from the main memory to the video memory. In addition, GPU-based methods will be further accelerated by future GPUs, because they are rapidly increasing the performance in terms of floating point operations and memory bandwidth [12].

Although GPUs are becoming flexible with enhancing programmability, their special-purpose architecture is still a barrier to achieve both fast decompression and high compression ratio. For example, random accesses are not fast compared with sequential accesses [13, 14], slowing down the processing speed against complex, irregular computation including branching instructions. Furthermore, GPUs currently do not have large video memory of more than 1.5 GB. Therefore, we need a hybrid method [10] that performs data decompression both on the CPU and GPU in a cooperative manner. This hybrid method, which is our main idea, can offload GPU cycles to the CPU without any performance loss if the storage device or GPU is kept as a performance bottleneck after offloading. That is, although this offloading increases the communication amount between the main memory and the video memory, it will not degrade the rendering performance if the increased amount does not limit the data throughput. In addition, the offloading should be applied to a sequence of operations that are suited to the CPU architecture rather than the GPU

architecture.

The objective of our work is to develop an efficient compression mechanism capable of accelerating out-of-core rendering of time-varying scalar data. To achieve this, we develop a thread-based decompression pipeline consisting of two stages, each processed on the CPU and GPU, respectively. We expand on our preliminary work [10] in this paper, describing how our pipeline mechanism accelerates the hybrid method for higher throughput. The difference to the earlier work [10] is the pipeline mechanism that hides decompression and rendering costs using a multi-core system. Our decompression pipeline requires a sequence of double-compressed data as the input, each consisting of three volumes at successive time steps. The double-compressed data is then decompressed first by the CPU, and then by the GPU in a pipeline fashion. Experimental results are also presented to demonstrate the effectiveness of the pipeline by using a desktop PC equipped with a multi-core CPU.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 gives a brief summary on volume texture compression (VTC) [15] and LZO compression [16], which are the basis of our decompression pipeline. Section 4 describes the details of the pipeline with the hybrid method. Section 5 shows experimental results evaluating the pipeline in terms of performance, scalability, and image quality. Section 6 concludes the paper with future work.

2 Related Work

There are many pipelined rendering systems [17–20] that overlap data loading and rendering stages to increase the rendering rate. Since these systems deal with raw data, data compression techniques are not integrated into them. On the other hand, many compression-based systems [3–9] are also proposed to increase the rendering performance. In these systems, only Bernardon et al. [9] uses a pipelined system that overlaps decompression with rendering. To the best of our knowledge, however, there is no work reporting on a pipelined rendering system that exploits both of the CPU and GPU for decompression. Thus, it is still not clear how effectively data compression techniques and pipeline mechanisms contribute to higher performance. Table 1 shows a classification of existing compression-based systems. As we mentioned earlier, they can be classified into three groups where data is decompressed: a hybrid method [10]; GPU-based methods [3–7]; and CPU-based methods [8, 9].

Most of prior systems use only the GPU for data decompression. Akiba et al. [3] propose a combination of two lossless compression techniques that exploit spatial coherence. Their system first applies the wavelet transformation [21] to each volume in order to convert them into a hierarchical represen-

tation. This wavelet-based compression allows users to select an appropriate resolution level according to interactivity and image quality. After this, the system applies a packing procedure to a selected subset of the low-resolution data. This procedure partitions the volume into 3-D blocks, which are then reduced by replacing the same blocks with pointers to the representative block. On the other hand, data decompression (unpacking) is done at run-time by a fragment program running on the GPU. However, this unpacking procedure could cause random accesses in the worst case, because it exploits the coherence across 3-D blocks. We think that such coherence should be exploited by the CPU, which has a larger cache with a higher tolerance to lower degrees of locality. Our system realizes this by performing LZO decompression on the CPU. Furthermore, the system exploits both temporal and spatial coherence, although it is a lossy compression.

Binotto et al. [4] also realize a lossless compression method. Their method partitions the volume into 3-D blocks, which are then reduced by replacing the same blocks with the representative block. In addition to this spatial coherence, temporal coherence is also exploited by replacing blocks between different time steps. Since this replacement requires an exact match between different blocks, their method is effective for highly sparse and temporally coherent data.

Fout et al. [5] realize lossy compression based on vector quantization. Their system has two modes depending on coherence to be exploited: spatial and temporal modes. In the spatial mode, their system packs $2 \times 2 \times 2$ neighboring voxels into a vector, and then applies quantization to the vector in order to approximate it with a representative vector. Schneider and Westermann [6] also use vector quantization but they exploit both spatial and temporal coherence by sharing representatives between successive time steps. Similar to Akiba's packing procedure, vector quantization produces representative vectors and pointers to them. Due to the nature of pointers, this sequence can result in irregular data, which generally decreases compression ratio. Therefore, when we combine multiple compression methods in a pipeline, such irregular data should not be generated at the first compression stage in order to avoid disturbing the second compression method. This also implies that complex decompression algorithms should be executed on the CPU rather than on the GPU.

Lum et al. [7] use the discrete cosine transform (DCT) to achieve lossy compression. Their hardware-assisted solution exploits temporal coherence by transforming data into a set of coefficients. These coefficients are then quantized to create a more compact representation, allowing us to discard coefficients with higher energy, which are not important in terms of image quality. Since the discard level can be selected by users, their system is capable of choosing the appropriate image quality.

A CPU-based method is proposed by Strengert et al. [8]. Their system is based on the wavelet-based compression [22] that exploits spatial coherence at each time step. It employs the wavelet transformation to generate a hierarchical octree, which is then compressed by an entropy encoding. During rendering, it decompresses data on the CPU and then transfers the raw data to the video memory through the graphics bus. Therefore, the performance of this method could be limited by the graphics bus especially if we increase the compression ratio (for higher performance).

Bernardon et al. [9] also use a CPU-based method but they overlap the decompression time with the rendering time. Their system uses vector quantization to exploit temporal coherence in unstructured grids. Although this system demonstrates successful overlaps, the decompression stage is processed only on the CPU. Accordingly, the raw data is transferred through the graphics bus. To avoid this, we think that the decompression stage itself should be structured into a pipeline.

Finally, a raw renderer is proposed by Furumura et al. [23] to deal with the problem of increasing data size. They develop a concurrent technique capable of processing simulation and visualization at the same time. This concurrent technique directly accesses the simulation results on the main memory, eliminating the need to store huge amounts of data. Therefore, data compression techniques are not needed for visualization. However, the simulation performance can limit the frame rate, because the simulation must be executed at every frame. In addition, the interactivity in the visualization phase might be reduced on batch-style high-performance computing systems, which are needed for the simulation.

3 Preliminaries

3.1 Volume Texture Compression (VTC)

VTC [15] is a lossy compression method that approximates voxel values by linear interpolation of representative values. This method is supported by nVIDIA GPUs [11] and is standardized as a part of OpenGL extensions [24]. One remarkable advantage of this method is fast data decompression accelerated by a hardware component on the GPU. Although VTC has four modes in terms of the format of input data, we only explain the COMPRESSED_RGB_S3TC_DXT1_EXT mode in this paper, which compresses 24-bit RGB data. We use this mode as an underlying method for our pipeline, because it has the highest compression ratio of 6:1.

Fig. 1 shows an overview of VTC. Given a volume in the RGB format, VTC partitions the volume into $4 \times 4 \times 1$ voxel blocks $B_1, B_2, \ldots, B_N$, which are then applied to an approximation algorithm to generate compressed data. This algorithm replaces each B_k ($1 \leq k \leq N$) of the blocks, namely 16 voxel values $v_{1,1}, v_{1,2}, \cdots, v_{4,4}$, with a compressed block B'_k containing (1) two representative values $R_k = \langle c_1, c_2 \rangle$ in 4 bytes and (2) an interpolation table $T_k = \{m_{i,j} \mid 1 \leq i \leq 4, 1 \leq j \leq 4\}$ in 4 bytes, where $m_{i,j}$ in 2 bits represents the approximation mode for voxel $v_{i,j}$. Since $m_{i,j}$ has 2-bits information, four approximation modes are predefined [15, 25]: c_1, c_2, $(2c_1 + c_2)/3$, and $(c_1 + 2c_2)/3$. For example, if the algorithm determines that the value for voxel $v_{i,j}$ should be approximated by $(2c_1 + c_2)/3$, then it stores $m_{i,j} = 2$ in table T_k, according to the predefined scheme. This linear interpolation is independently applied to every block, so that each block has a different table T_k. Therefore, VTC exploits spatial coherence in a small $4 \times 4 \times 1$ block.

In contrast to the encoding scheme mentioned above, the decoding scheme returns the approximated value by linear interpolation of two representative values c_1 and c_2. For example, it returns $(2c_1 + c_2)/3$ as the value of voxel $v_{i,j}$ if $m_{i,j} = 2$. Note that this interpolation is quickly done by the on-the-fly decoder implemented as a hardware component. In addition, the on-the-fly decompression does not require additional memory to store the decompressed data, because the entire volume is not decompressed at a time. The GPU also exploits two parallelisms to further accelerate this decompression: (1) single instruction multiple data (SIMD) instructions [26] and (2) vector instructions, allowing us to simultaneously process multiple voxels in a volume and multiple RGB channels in a single voxel, respectively. As compared with SIMD instructions, vector instructions are useful to increase the data granularity, which contributes to the full utilization of memory bandwidth.

3.2 Lempel-Ziv-Oberhumer (LZO) Compression

LZO [16] is a real-time data decompression/compression library based on a lossless compression algorithm. This library uses a dictionary coder [27], which scans the input data to replace repeating data with references back to the original data. During this scan, a sliding window keeps recent scanned data to perform a replacement operation when a longest match is found in the window. Therefore, LZO obtains high compression ratio if the same data sequence frequently appears in the window. Similar to VTC, LZO has multiple compression modes. In this paper, LZO denotes LZO1X-1, which is regarded as the fastest mode in most cases.

Note that the LZO library provides fast decompression, because it does not require complex operations to decode data. It simply replicates the original

data to places where the references exist. In addition, this replication can be quickly done if the sliding window is stored entirely in a CPU cache. Actually, the LZO library has a 64-KB window, which can be stored in an L2 cache. In addition to this cache optimization, the LZO library has a hash table for search acceleration.

4 Decompression Pipeline

We now describe our decompression pipeline designed on the basis of compression methods presented in Section 3. We first show the design aspects of our pipeline, and then describe the hybrid method [10] and the pipeline mechanism.

Fig. 2 shows an overview of the hybrid method. As we mentioned in Section 1, the key idea of this method is to perform data decompression both on the CPU and the GPU. To realize this, we combine two different compression methods, M_c and M_g, each running on the CPU and GPU, respectively. Therefore, the raw data must be converted into double-compressed data in advance of rendering. To do this, the hybrid method compresses the raw data first by M_g, and then by M_c. During the visualization phase, this double-compressed data is decompressed by applying the decompression methods in the opposite order: first by the CPU, and then by the GPU.

We also develop a pipeline mechanism in order to overlap data loading, data decompression, and rendering stages. This mechanism increases rendering throughput, because it allows us to process successive data simultaneously at different pipeline stages. In addition, the pipeline mechanism contributes to hide the decompression overhead incurred on the CPU. In our hybrid method, this overhead is due to the cycles offloaded from the GPU. Therefore, we think that the pipeline mechanism is essential to maximize the effectiveness of the hybrid method.

4.1 Design Aspects

Compression methods M_c and M_g must satisfy the following three requirements to achieve fast volume rendering of time-varying data.

R1. Both M_c and M_g achieve a compression ratio sufficiently high to pay decompression cost T_2, where T_2 represents the time for data decompression.
R2. M_c and M_g exploit different coherences.

R3. M_g outputs a sequence of data such that it keeps the coherence that will be exploited later by M_c.

The first requirement R1 is a general constraint to compression methods. This requirement implies that there is a tradeoff relation between compression ratio and decompression time. From this viewpoint, although a high-compression method significantly reduces the loading time T_1, it is not satisfactory if it fails to reduce the total time $T_1 + T_2$.

On the other hand, the remaining requirements R2 and R3 are inherent in hybrid (two-stage) compression methods. Requirement R2 indicates that it is not effective if similar compression methods are repeatedly applied to the volume data. For example, a combination of a temporal encoder and a spatial encoder satisfies this requirement, because they focus on independent coherences. Requirement R3 comes from the nature of sequential execution of methods M_c and M_g. For example, M_c and M_g are not a good pair if M_g destroys the coherence which M_c exploits later in the second compression stage.

4.2 Hybrid Compression

To satisfy requirements R1, R2, and R3, our hybrid method uses packed volume texture compression (PVTC) [10] for M_g and LZO [16] for M_c. PVTC is a lossy compression method for time-varying volume data containing scalar values. It extends VTC [15] to exploit temporal coherence in time-varying data. Using PVTC and LZO, the hybrid method iteratively compresses three successive volumes from the beginning of volume sequence. In the following discussion, let V_t denote the volume at time step t, where $0 \leq t < t_{max}$. Let $V_t(x, y, z)$ denote the voxel value of point (x, y, z) in volume V_t. Then, the compression procedure can be written as follows.

(1) $t \leftarrow 0$.
(2) Data packing. Fig. 3 illustrates this step. Given three successive raw volumes V_t, V_{t+1}, and V_{t+2}, the method packs them into RGB data to obtain a packed volume $\mathbf{C}_{\lfloor t/3 \rfloor}$. To do this, three scalar values $V_t(x, y, z)$, $V_{t+1}(x, y, z)$, and $V_{t+2}(x, y, z)$ in the same location are copied to R, G, and B channels of $\mathbf{C}_{\lfloor t/3 \rfloor}(x, y, z)$, respectively. Here, $\mathbf{C}_{\lfloor t/3 \rfloor}(x, y, z)$ represents the voxel value of point (x, y, z) in $\mathbf{C}_{\lfloor t/3 \rfloor}$. Empty values are padded if $t_{max} \bmod 3 \neq 0$.
(3) Data compression by VTC. The packed volume $\mathbf{C}_{\lfloor t/3 \rfloor}$ is given to a VTC encoder to obtain a compressed texture $\mathbf{X}_{\lfloor t/3 \rfloor}$.
(4) Data compression by LZO. The compressed texture $\mathbf{X}_{\lfloor t/3 \rfloor}$ is given to a LZO encoder to obtain a double-compressed texture $\mathbf{Z}_{\lfloor t/3 \rfloor}$.
(5) $t \leftarrow t + 3$. Goto step (2) if $t < t_{max}$.

The combination of PVTC and LZO satisfies requirement R2, because PVTC and LZO focus on different coherences. That is, PVTC exploits temporal and spatial coherence in a small $4 \times 4 \times 1$ block. On the other hand, LZO exploits spatial coherence across small blocks, because it receives PVTC compressed textures as inputs. Here, the sliding window in the LZO library covers 8192 VTC compressed blocks at a time, which is equivalent to $512 \times 256 \times 1$ RGB voxels. Therefore, LZO compression can be regarded as a slice-based compression if a volume consists of 512×512 voxel slices. In this sense, LZO and PVTC do not exploit exactly the same coherence, though they exploit spatial coherence.

With respect to requirement R3, this requirement can be satisfied as follows.

- PVTC can keep the coherence between different blocks in the input volume, because it will encode similar blocks into compressed blocks with similar representative values and interpolation tables. Thus, the coherence between blocks in the raw volume is also kept in compressed data for later LZO compression.
- The coherence mentioned above is expressed in the format of compressed data, because PVTC produces fixed-size data for each block in first-in, first-out (FIFO) order.

From the viewpoint of architectural design, our combination aims at achieving fast decompression by taking advantage of GPU and CPU architectures. For example, as we mentioned in Section 3.1, the GPU has hardware acceleration capabilities which the CPU does not have. On the other hand, the CPU has a larger cache as compared with the texture cache on the GPU. This larger cache is suited to duplicating operations in LZO decompression, because these operations refer to a large area in the sliding window. Actually, the cache working set in an nVIDIA GeForce 8800 GTX card is 8 KB per multiprocessor [28], and thus the entire window cannot be stored in the texture cache. In summary, the CPU cooperates with the GPU by providing a larger cache to accelerate decompression for large area.

4.3 Pipelined Data Decompression

Once the raw data is converted into double-compressed textures, our pipelined renderer repeatedly decompresses them during visualization. Note that PVTC allows us to load data at every three time steps, because each of PVTC compressed textures contains time-series data. The decompression scheme can be written as follows (see also Fig. 4).

S0. $t \leftarrow 0$.

S1. Data loading from the storage device. A double-compressed texture $\mathbf{Z}_{\lfloor t/3 \rfloor}$ is transferred from the hard disk to the main memory.
S2. Data decompression by LZO. The CPU decompresses $\mathbf{Z}_{\lfloor t/3 \rfloor}$ to obtain a compressed texture $\mathbf{X}_{\lfloor t/3 \rfloor}$.
S3. Texture-based volume rendering [29,30] on the GPU.
  (a) Texture transfer. $\mathbf{X}_{\lfloor t/3 \rfloor}$ is transferred from the main memory to the video memory.
  (b) Data decompression by PVTC. For all voxels (x, y, z), the approximated value $\tilde{V}_t(x, y, z)$ is automatically obtained by simply accessing the appropriate channel of packed voxel $\mathbf{C}_{\lfloor t/3 \rfloor}(x, y, z)$. That is, the VTC decoder returns R, G, or B channel data if t mod 3 = 0, 1, or 2, respectively. Note that this branching instruction can be eliminated from fragment programs. Instead of fragment processors, the CPU performs this by binding the appropriate fragment program that refers one of R, G, or B channel data.
  (c) Data classification by a transfer function. The final color and opacity of $\tilde{V}_t(x, y, z)$ are determined by a transfer function.
  (d) $t \leftarrow t + 1$. Goto step (b) if t mod $3 \neq 0$.
S4. $t \leftarrow 0$ if $t \geq t_{max}$. Goto stage S1.

The above stages S1, S2, and S3 are overlapped by our pipeline mechanism, as shown in Fig. 4. To realize this mechanism by software, we use the POSIX thread library [31]. Fig. 5 shows a pseudocode of the mechanism. We create three threads, each responsible for processing one of the three stages. This mechanism is thread-safe if it satisfies the following conditions C1 and C2 during execution.

C1. For all time steps t, where $0 \leq t < t_{max}$, the volume at time step t is processed sequentially from stage S1 to stage S3.
C2. The rendering thread produces images in an ascending order of time step t.

To satisfy condition C1, we create threads such that each of the threads is blocked with pthread_cond_wait() until it receives a signal from the previous thread in the pipeline. This previous thread, on the other hand, sends a signal to the next thread when it finishes an incoming task. In addition, the rendering thread sends a wake-up signal to the loading thread after completing a rendering task.

Condition C2 is essential to guarantee that the stream data cannot pass each other. If we lack this condition, we cannot build a correct pipeline on threads. This condition can be satisfied by a FIFO policy. To realize this, each thread has variable "tstep," which stores the latest time step processed at the corresponding stage. This information is then used to prevent the stream data from overtaking each other. That is, a thread is allowed to process the data if

it has already been processed by the previous threads. Otherwise, the thread will be blocked with pthread_cond_wait() placed in a while loop.

Finally, once the time-varying data is sent to the GPU, the final image will be quickly generated by the GPU. We employ a texture-based rendering method [29,30], which is fully accelerated by hardware components in the GPU, such as texture mapping and alpha blending hardware. We currently use 3-D textures rather than 2-D textures, because 2-D textures consume three times more memory [30] needed for a copy of the dataset for each orientation. Although this might be a trivial problem for rendering of non-time-varying data, it is critical for out-of-core (data-intensive) rendering.

5 Experimental Results

In order to evaluate the proposed pipeline in terms of performance, scalability, and image quality, we implemented it using the C++ language, the OpenGL library [24], the Cg toolkit [32], and the POSIX thread library [31]. For experiments, we use a desktop PC equipped with 2 GB of main memory and a 320GB SATA disk device. This PC has a Core 2 Quad CPU running at 2.66 GHz clock speed and an nVIDIA Quadro FX 4600 card having 768 MB of video memory. The graphics card is connected to a PCI Express 16X bus. The experiments are performed using Windows XP with driver version 162.55.

Table 2 summarizes six datasets D1–D6 used for experiments. Datasets D4, D5, and D6 are high-resolution versions of datasets D1, D2, and D3, respectively. See also Fig. 6 for their visualization results. Datasets D1 and D2 [33] are fluid dynamics data showing a turbulent jet and a turbulent vortex flow, respectively. Both datasets have low temporal coherence due to moving objects. However, dataset D1 has high spatial coherence at any time step, because it consists of many transparent voxels. Dataset D3 shows a sequence of lung deformations. It represents a deformation process of nonrigid registration [34, 35] that aligns a lung dataset to another dataset obtained at a different time. These datasets are slightly different from each other due to breathing. Since the deformations are small, this dataset has high temporal coherence. All datasets have voxels of 1-byte scalar data.

Table 2 also shows the compression ratio for each dataset. As we mentioned in Section 4.2, PVTC is a lossy compression based on VTC, which has the compression ratio of 6:1. On the other hand, the combination of PVTC and LZO compression contributes to further reduction of the data size: the highest ratio of 12:1 and the lowest ratio of 1.1:1, each obtained by LZO compression. Thus, the total compression ratio ranges from 6.5:1 to 71.7:1 after LZO compression. However, LZO compression is not effective for dataset D2. The coherence in

this dataset is not high, so that the match found in the sliding window is not long enough to obtain a high compression ratio. Actually, the compression ratio is at most 1.1:1 even if we apply LZO compression directly to D2, without PVTC. This indicates that the data originally lacks spatial coherence, thus, we think that the compression ratio of 6.5:1 is not due to PVTC, which keeps coherence that will be exploited by LZO compression. We also think that hardware supported compression methods such as PVTC, even though they are lossy, are essential to achieve fast rendering for low-coherence data.

5.1 Rendering Performance

To evaluate the performance gain of the proposed pipeline, we now compare it with three methods: (1) a straightforward method that transfers raw data from the storage device to the video memory; (2) a GPU-based method that uses PVTC on the GPU but no compression method on the CPU; and (3) a hybrid method [10] that is not structured in a pipeline.

Fig. 7 shows average frame rates for all methods. Datasets are rendered on a screen with an appropriate size: a 256×256 pixel screen for D1 and D2; a 512×512 pixel screen for D3, D4, and D5; and a 1024×1024 pixel screen for D6. The viewing direction is initially set to z-axis direction, and then it is rotated 2 degrees around x- and y-axes when the time step is updated. For every measurement, we rebooted the machine so that disk cashes are cleaned before measurement. Thus, the frame rates in our experiments are mainly determined by the compression ratio and the volume size rather than the number t_{max} of time steps (see also Section 5.2).

Our pipelined hybrid method achieves a video rate of 41 frames per second (fps) for dataset D4: $258 \times 258 \times 208$ voxel data with 150 time steps. It also demonstrates an almost interactive rate of 8 fps for the largest dataset D6: $512 \times 512 \times 295$ voxel data with 411 time steps. As compared with prior results [3, 10, 36], these performance results are obtained using more recent hardware with more CPUs, but we think that our results are reasonable in performance. For example, Akiba et al. [3] achieves 1.1 fps for $256 \times 256 \times 1024$ voxel data with 400 time steps. Fout et al. [36] reports approximately 6 fps for non-time-varying data of $512 \times 512 \times 512$ voxels.

For all datasets, our pipeline mechanism achieves 33–122% improvement over the hybrid method, which achieves 3.3–7.4 times higher rendering performance than the raw renderer. The pipeline also achieves 62–201% improvement over the GPU-based method. One interesting point here is that the hybrid method fails to improve the performance for dataset D2, as compared with the GPU-based method. This means that LZO compression fails to obtain higher com-

pression ratio due to the lack of higher coherence. Actually, Table 2 shows that the total compression ratio of 6.5:1 is mostly obtained by PVTC. Despite this fact, the pipeline mechanism increases the rendering rate from 66 fps to 111 fps. Therefore, we think that the pipeline mechanism is essential to maximize the timing advantage of the hybrid method.

Fig. 8 shows the average execution time T and its breakdown normalized to a single time-step data: (1) average time T_1' spent for data loading at stage S1; (2) average time T_2' spent for LZO decompression at stage S2; (3) average time T_3' spent for texture transfers at stage S3(a); and (4) average time T_4' spent for texture-based rendering at stage S3 except texture transfers. Note that the execution time T for the pipelined hybrid method does not equal to the sum of breakdown $T_1' \ldots T_4'$, because each breakdown is measured concurrently on the corresponding thread.

We first compare the pipelined hybrid method with the hybrid method to analyze the effects of the pipeline mechanism. In the hybrid method, overheads $T_1' \ldots T_4'$ account for most of the entire time T. In contrast, the pipelined hybrid method overlaps these four overheads, reducing execution time T. Actually, we can see that time T is smaller than the total of time $T_1' \ldots T_4'$ in Fig. 8. Thus, this overlap explains the 33–122% improvement mentioned before. In many cases, the pipeline mechanism reduces the entire time T roughly to time T_1', because data loading operations at stage S1 is usually a performance bottleneck in the pipeline. Therefore, the decompression cost T_2' is usually hidden by time T_1' in our pipeline. Thus, the pipeline mechanism is capable of hiding the overheads of the hybrid method by overlapping overhead T_2' with other overheads. It also should be noted that the thread-related overheads are much smaller than the entire time T. The overheads can be estimated by comparing time $T_1' + T_2' + T_3' + T_4'$ between the pipelined hybrid method and the hybrid method.

We next analyze the effects of data compression methods. In Fig. 8, we can see that the GPU-based method achieves 3.0–3.8 times higher performance as compared with the straightforward method. This improvement is achieved by the reduction of data size, which decreases loading time T_1'. Due to the same reason, the hybrid method also achieves 1.2–2.0 times higher performance than the GPU-based method, for all datasets except D2. For dataset D2, the hybrid method fails to show the performance gain over the GPU-based method, due to the lower compression ratio. Therefore, the hybrid method slightly reduces loading time T_1', but it has longer execution time T due to additional decompression overhead T_2'.

Fig. 8 also indicates that most of execution time T is spent for loading operations if we do not use compression methods. Therefore, overlapping the loading stage with other stages is not so effective in this case. Thus, we must

reduce loading time T_1' by using compression methods in order to maximize the effectiveness of the pipeline mechanism. Note that loading time T_1' is mainly limited by the seek time of the storage device if we render small datasets, such as D1 and D2. Actually, the seek time of our storage device (Western Digital WD3200AAKS) is 8.9 ms, which is close to loading time T_1' (approximately 10 ms) in Fig. 8.

We next analyze the behavior of the pipelined hybrid method. Fig. 9 shows a timeline chart explaining how the pipeline renders dataset D5 on the multi-core PC. This chart is generated from a trace file obtained from an execution of an instrumented program. Although the pipeline runs sequentially for the first three time steps, it is fully parallelized after time step 6. Thus, three threads are running simultaneously on a multi-core CPU, showing an effective utilization of computing resources. We think that such a parallel implementation is important to realize a full utilization in future many-core machines.

Finally, Fig. 10 presents the time-varying properties of the pipelined hybrid method in terms of the compression ratio and the rendering performance. It shows that the first time step is rendered at approximately 10 fps, due to the lack of pipeline effects (latency hiding). Since our pipeline transfers a collection of three time steps at a time, the rendering performance decreases when the rendering thread loads a compressed texture from the main memory. Fig. 10 shows that this periodical behavior actually occurs at every three time steps. To resolve this problem, we must split the rendering thread into two different threads: one for texture transfer at stage S3(a); and the other for the remaining stages S3(b)–(d). This modification will improve the performance on a quad-core machine, because it allows us to prefetch a compressed texture while rendering the current volume. Thus, we can overlap time T_3' with T_4'.

With respect to the compression ratio, it shows an irregular behavior in Fig. 10(a), but the rendering performance shows a periodical behavior. This means that the pipeline mechanism successfully hides the irregularity at the loading and decoding stages. However, our pipeline cannot hide the overhead incurred at the rendering stage, namely the last stage in the pipeline. For example, we observe that the cgGLBindProgram() function called for program switching by the rendering thread spends longer time every after two successive calls of the function. This explains why the rendering performance decreases at time step t, as compared to time step $t-1$, where $t \bmod 3 = 2$. Due to the same reason, the entire performance can vary according to the viewing point and direction.

We first analyze the scalability of our system in terms of the capacity. Let n be the volume size. The system assumes that the video memory is large enough to store a compressed texture $\mathbf{X}_{\lfloor t/3 \rfloor}$, where $0 \leq t < t_{max}$. Since $\mathbf{X}_{\lfloor t/3 \rfloor}$ is generated from three raw volumes V_t, V_{t+1}, and V_{t+2} with a compression ratio of 6:1, a compressed texture requires $n^3/2$ memory space. Therefore, we need at least 512 MB of video memory to render $1024 \times 1024 \times 1024$ voxel data. In other words, 512 MB of video memory is not sufficient to render such data, because it lacks the memory space for frame buffer. Thus, we think that the maximum cubic size is limited by $1024 \times 1024 \times 1024$ voxels on current graphics cards.

We next analyze how the rendering performance is affected by the multi-level storage hierarchy consisting of the disk device, main memory, and video memory. Let σ be the compression ratio obtained by the hybrid method. Let $s = n^3/2\tau$ be the file size of a double-compressed data, where $\tau = \sigma/6$ represents the compression ratio obtained by LZO. Let F be the average frame rate for a single time step. Since the bottleneck stage in the pipeline limits the frame rate, the frame rate is given by:

$$F = 1/max(T'_1, T'_2, T'_3 + T'_4). \tag{1}$$

Each of times $T'_1 \ldots T'_4$ can be simply modeled as follows: the loading thread receives $s/3$ data in average from the disk device; the decoding thread receives the $s/3$ data and then sends $s\tau/3$ data; the rendering thread receives the $s\tau/3$ data and then loads $O(n^3)$ single-precision data to produce the image. We assume here that the rendering cost is proportional to the number of voxels in a raw volume. Therefore, we obtain

$$T'_1 = s/3B_1, \tag{2}$$
$$T'_2 = s(1+\tau)/3B_2, \tag{3}$$
$$T'_3 = s\tau/3B_3, \tag{4}$$
$$T'_4 = 2Ks\tau/B_4, \tag{5}$$

where K represents the coefficient of the rendering cost and $B_1 \ldots B_4$ represent the throughput of data loading, that of LZO decoding, that of texture transfer, and that of rendering, respectively. Equations (2)–(5) indicate that the volume size n does not determine the bottleneck stage in the pipeline. The bottleneck stage is determined by the compression ratio τ of LZO and by the throughputs $B_1 \ldots B_4$.

The throughputs can be estimated by using the measured values of compres-

sion ratio and execution time presented in Table 2 and Fig. 8, respectively. According to this estimation, the lowest throughputs we obtained are as follows: $B_1 = 13$; $B_2 = 452$; $B_3 = 205$; and $B_4 = 1352K$. Using these measured throughputs and Equations (2)–(5), we obtain the following relations:

$$T'_1 > T'_2 \geq T'_3 + T'_4, \quad \text{if } \tau \leq 0.3, \tag{6}$$
$$T'_1 \geq T'_3 + T'_4 > T'_2, \quad \text{if } 0.3 < \tau \leq 8.3, \tag{7}$$
$$T'_3 + T'_4 > T'_1 > T'_2, \quad \text{if } 8.3 < \tau < 33.8, \tag{8}$$
$$T'_3 + T'_4 > T'_2 \geq T'_1, \quad \text{otherwise.} \tag{9}$$

In summary, the loading thread or the rendering thread can be a performance bottleneck in our desktop PC. The rendering thread determines the frame rate F when the compression ratio σ is higher than 49.8:1. Otherwise, the loading thread determines the frame rate F.

Actually, Figs. 8(c) and (f) show that the rendering thread becomes a performance bottleneck when viewing datasets D3 ($\sigma = 67.0$) or D6 ($\sigma = 71.7$). In particular, this bottleneck is critical if we render higher resolution data such as $512 \times 512 \times 512$ voxel data. Therefore, we think that $512 \times 512 \times 512$ voxel data is the maximum volume size that can be interactively rendered by our current system. We also think that this size will be increased by future GPUs, which achieve shorter rendering time T'_4 and transfer time T'_3 by a higher memory bandwidth and by a new graphics bus, respectively.

5.3 Image Quality

In order to evaluate the quality of rendered images, we measure PSNR (peak signal-to-noise ratio) values and analyze the error distribution. A higher PSNR value here represents higher quality with less noise. In general, PSNR values of at least 30 dB are desired for rendered images. As a rule of thumb, a PSNR value over 40 dB indicates that the image is very close to the original and a value between 30 to 40 dB means that the distortion is visible but acceptable [37].

Fig. 11 shows PSNR values measured for each dataset. For all datasets, we observe PSNR values of more than 33 dB. These results are competitive to prior work [7, 36] that achieves similar quality ranging from 28 to 49 dB. Fig. 12 shows the breakdown of pixel errors in produced images. It represents how exactly the hybrid method produces the original pixels. In most cases, more than 95% of pixels have an error within pixel value 3, as compared with the exact image produced from raw data. We think that this distribution is acceptable because pixel values are in the range [0,255].

Fig. 13 shows two pairs of images, each produced from the raw data and from the PVTC compressed data. Fig. 14 shows subtraction images produced from each pair. By comparing them in zoomed views, we can see some artifacts in Fig. 13(b) and Fig. 13(d), especially where neighboring pixels do not have similar values. More precisely, we can see a glow effect or rim in the lower right of the zoomed subimage in Fig. 13(b), where this is not a case in Fig. 13(a). This artifact could mislead the user into a different interpretation of simulation results when the original data is not available for comparison. Nevertheless, these artifacts do not significantly change the overall appearance of produced images. Thus, we think that the image quality of PVTC is tolerable for time-series analysis.

6 Conclusion

We have presented a decompression pipeline for accelerating volume rendering of time-varying scalar data. The proposed pipeline is based on a hybrid compression method and a thread-based pipeline mechanism. The hybrid method uses a combination of two different compression methods, namely PVTC and LZO compression, each running on the GPU and on the CPU. Our combination is designed to exploit both temporal and spatial coherence in time-varying data. It also makes a good use of hardware components on the GPU and a large cache on the CPU. The pipeline mechanism overlaps data loading, data decompression, and rendering stages, in order to increase the rendering rate.

Experimental results show that our pipelined method achieves a video rate of 41 fps in average for $258 \times 258 \times 208$ voxel data with 150 time steps. It also demonstrates an almost interactive rate of 8 fps for $512 \times 512 \times 295$ voxel data with 411 time steps. Our pipeline mechanism achieves 33–122% improvement over the nonpipelined hybrid method, which achieves 3.3–7.4 times higher rendering performance than the raw renderer. We also find that most of the execution time is spent for loading operations if we do not use compression methods. Therefore, we think that loading time must be reduced by compression methods in order to maximize the performance benefit of the pipeline mechanism. Furthermore, the pipeline mechanism is also useful to hide the decompression overhead inherent in multi-stage compression methods.

One future work is to develop an adaptive framework for visualization of higher resolution data. Although our compression pipeline provides an effective solution to visualization of large time-step data, it assumes that the capacity of video memory is large enough to store the entire of a compressed texture. A scalable, parallelization scheme [38] will be useful to reduce the memory requirement for a graphics card. We also think that a multiresolution data structure [39] contributes to progressive visualization that increases the inter-

activity for large volume.

Acknowledgments

This work was partly supported by MEXT Grant-in-Aid for Scientific Research for Young Researchers (19700061), JSPS Grant-in-Aid for Scientific Research (B)(2)(18300009), and the Global COE Program "in silico medicine" at Osaka University. We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] M. Levoy, Display of surfaces from volume data, IEEE Computer Graphics and Applications 8 (3) (1988) 29–37.

[2] K.-L. Ma, Visualizing time-varying volume data, Computing in Science and Engineering 5 (2) (2003) 34–42.

[3] H. Akiba, K.-L. Ma, J. Clyne, End-to-end data reduction and hardware accelerated rendering techniques for visualizing time-varying non-uniform grid volume data, in: Proc. 4th Int'l Workshop Volume Graphics (VG'05), 2005, pp. 31–39.

[4] A. P. Binotto, J. L. Comba, C. M. Freitas, Real-time volume rendering of time-varying data using a fragment-shader compression approach, in: Proc. 6th IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG'03), 2003, pp. 69–76.

[5] N. Fout, K.-L. Ma, J. Ahrens, Time-varying, multivariate volume data reduction, in: Proc. 20th ACM Symp. Applied Computing (SAC'05), 2005, pp. 1224–1230.

[6] J. Schneider, R. Westermann, Compression domain volume rendering, in: Proc. 14th IEEE Visualization Conf. (VIS'03), 2003, pp. 293–300.

[7] E. B. Lum, K.-L. Ma, J. Clyne, A hardware-assisted scalable solution for interactive volume rendering of time-varying data, IEEE Trans. Visualization and Computer Graphics 8 (3) (2002) 286–301.

[8] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, T. Ertl, Large volume visualization of compressed time-dependent datasets on GPU clusters, Parallel Computing 31 (2) (2005) 205–219.

[9] F. F. Bernardon, S. P. Callahan, J. L. Comba, C. T. Silva, An adaptive framework for visualizing unstructured grids with time-varying scalar fields, Parallel Computing 33 (6) (2007) 391–405.

[10] D. Nagayasu, F. Ino, K. Hagihara, Two-stage compression for fast volume rendering of time-varying scalar data, in: Proc. 4th Int'l Conf. Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE'06), 2006, pp. 275–284.

[11] J. Montrym, H. Moreton, The GeForce 6800, IEEE Micro 25 (2) (2005) 41–51.

[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.

[13] I. Buck, K. Fatahalian, P. Hanrahan, GPUBench: Evaluating GPU performance for numerical and scientific applications, in: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04), 2004, p. C-20.

[14] B. He, N. K. Govindaraju, Q. Luo, B. Smith, Efficient gather and scatter operations on graphics processors, in: Proc. High Performance Networking and Computing Conf. (SC'07), 2007, 12 pages (CD-ROM).

[15] OpenGL Extension Registry, Gl_nv_texture_compression_vtc, `http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_compression_vtc.txt` (2004).

[16] M.F.X.J. Oberhumer, LZO real-time data compression library, `http://www.oberhumer.com/opensource/lzo/` (Oct. 2005).

[17] H. Yu, K.-L. Ma, A study of I/O methods for parallel visualization of large-scale data, Parallel Computing 31 (2) (2005) 167–183.

[18] T. Chiueh, K.-L. Ma, A parallel pipelined renderer for time-varying volume data, in: Proc. 2nd Int'l Symp. Parallel Architectures, Algorithms and Networks (I-SPAN'97), 1997, pp. 9–15.

[19] W. Bethel, B. Tierney, J. Lee, D. Gunter, S. Lau, Using high-speed WANs and network data caches to enable remote and distributed visualization, in: Proc. High Performance Networking and Computing Conf. (SC'00), 2000.

[20] X. Cavin, C. Mion, A. Filbois, COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation, in: Proc. 16th IEEE Visualization Conf. (VIS'05), 2005, 8 pages (CD-ROM).

[21] J. Clyne, The multiresolution toolkit: Progressive access for regular gridded data, in: Proc. 3rd IASTED Int'l Conf. Visualization, Imaging, and Image Processing (VIIP'03), 2003, pp. 152–157.

[22] S. Guthe, M. Wand, J. Gonser, W. Straßer, Interactive rendering of large volume data sets, in: Proc. 13th IEEE Visualization Conf. (VIS'02), 2002, pp. 53–60.

[23] T. Furumura, L. Chen, Parallel simulation of strong ground motions during recent and historical damaging earthquakes in tokyo, japan, Parallel Computing 31 (2) (2005) 149–165.

[24] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, 5th Edition, Addison-Wesley, Reading, MA, 2005.

[25] OpenGL Extension Registry, Gl_ext_texture_compression_dxt1, http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_dxt1.txt (2004).

[26] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, 2nd Edition, Addison-Wesley, Reading, MA, 2003.

[27] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Information Theory 23 (3) (1977) 337–343.

[28] nVIDIA Corporation, CUDA Programming Guide Version 1.1, http://developer.nvidia.com/cuda/ (Nov. 2007).

[29] B. Cabral, N. Cam, J. Foran, Accelerated volume rendering and tomographic reconstruction using texture mapping hardware, in: Proc. 4th Symp. Volume Visualization (VVS'94), 1994, pp. 91–98.

[30] M. Hadwiger, J. M. Kniss, K. Engel, C. Rezk-Salama, High-quality volume graphics on consumer PC hardware, in: SIGGRAPH 2002, Course Notes 42, 2002.

[31] B. Nichols, B. Buttlar, J. P. Farrell, Pthreads Programming, O'Reilly & Associates, Newton, MA, 1996.

[32] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: A system for programming graphics hardware in a C-like language, ACM Trans. Graphics 22 (3) (2003) 896–897.

[33] K.-L. Ma, Time-Varying Volume Data Repository, http://www.cs.ucdavis.edu/~ma/ITR/tvdr.html (2003).

[34] J. V. Hajnal, D. L. Hill, D. J. Hawkes (Eds.), Medical Image Registration, CRC Press, Boca Raton, FL, 2001.

[35] F. Ino, K. Ooyama, K. Hagihara, A data distributed parallel algorithm for nonrigid image registration, Parallel Computing 31 (1) (2005) 19–43.

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

[37] Y. Wang, J. Ostermann, Y.-Q. Zhang, Video Processing and Communications, Prentice Hall PTR, Upper Saddle River, NJ, 2001.

[38] H. Childs, M. Duchaineau, K.-L. Ma, A scalable, hybrid scheme for volume rendering massive data sets, in: Proc. 6th Eurographics Symp. Parallel Graphics and Visualization (EGPGV'06), 2006, pp. 153–162.

[39] E. LaMar, B. Harmann, K. I. Joy, Multiresolution techniques for interactive texture-based volume visualization, in: Proc. 10th IEEE Visualization Conf. (VIS'99), 1999, pp. 355–361.

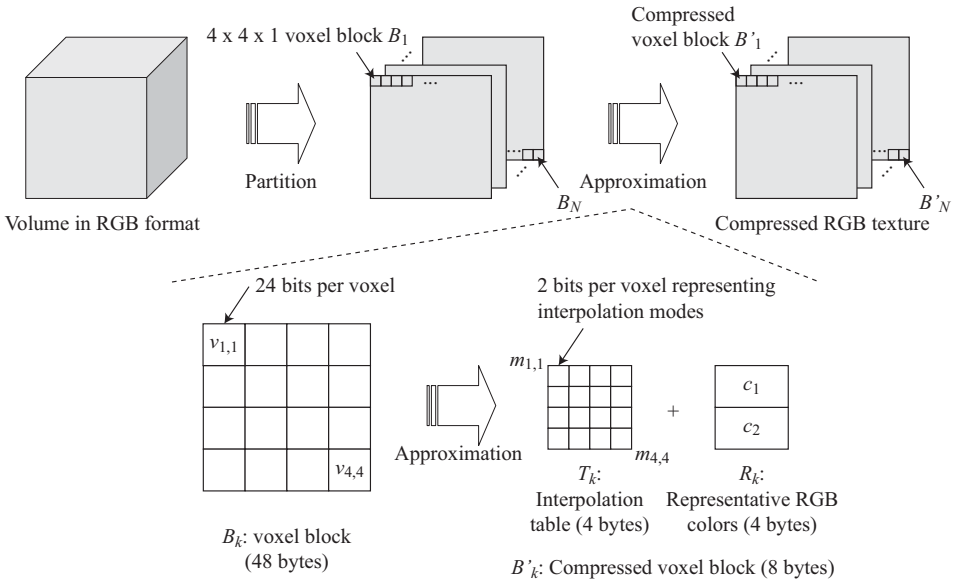


Fig. 1. Volume texture compression (VTC).

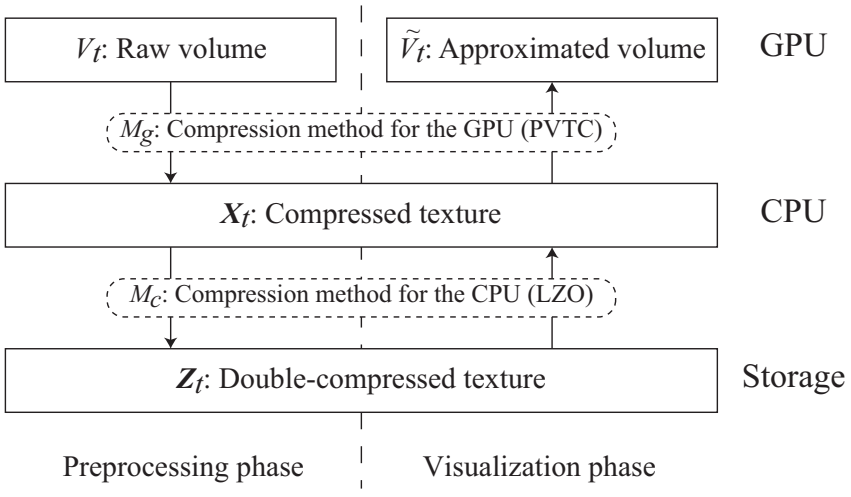


Fig. 2. Overview of hybrid compression method. This method uses two different compression methods, M_c and M_g, each running on the CPU and the GPU, respectively. Note that raw data must be compressed in advance of visualization.

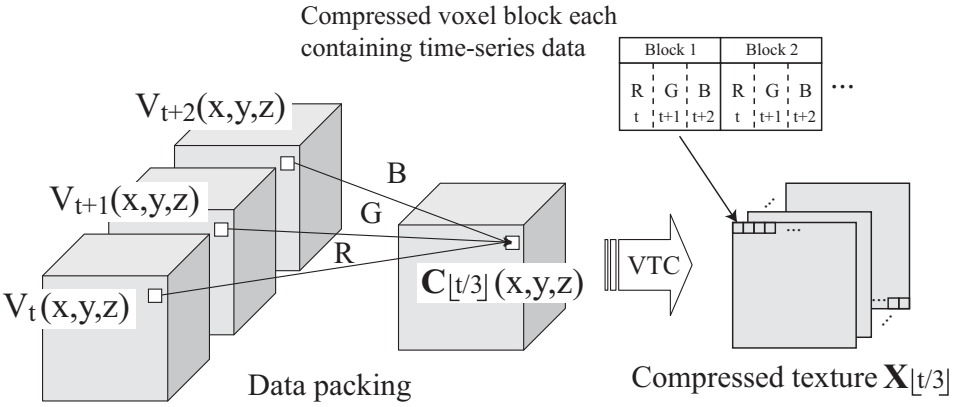


Fig. 3. Packed volume texture compression (PVTC). Time-series scalar voxels in the same location are packed into an RGB voxel. The packed RGB data is then compressed using VTC, which generates a sequence of compressed blocks, each containing time-series data.

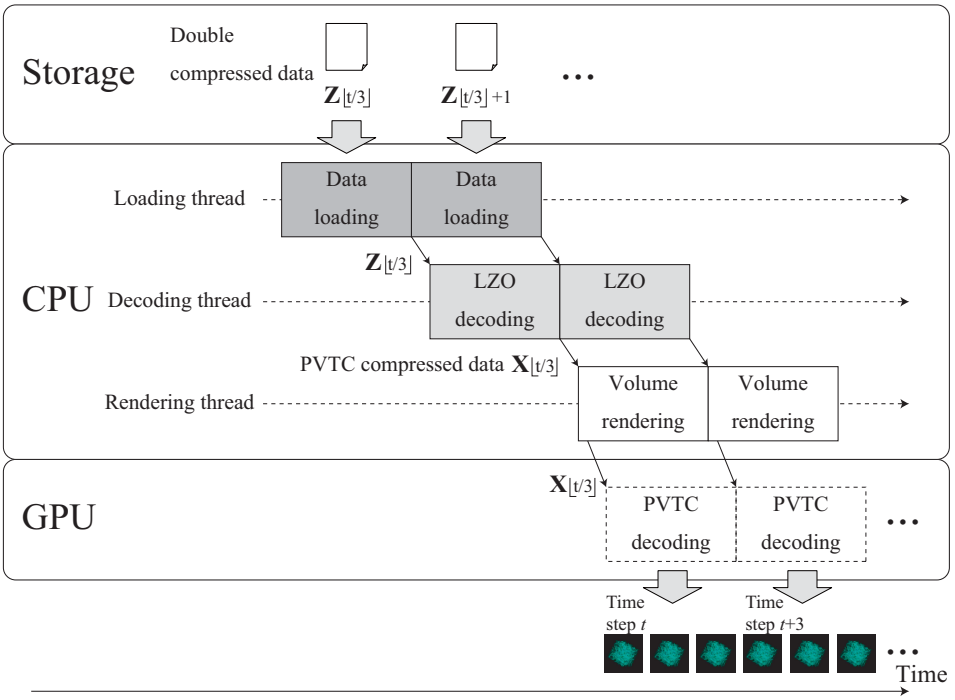


Fig. 4. Decompression pipeline implemented using multiple threads running on a multi-core CPU. The pipeline consists of three stages, each responsible for data loading, LZO decoding, and volume rendering.

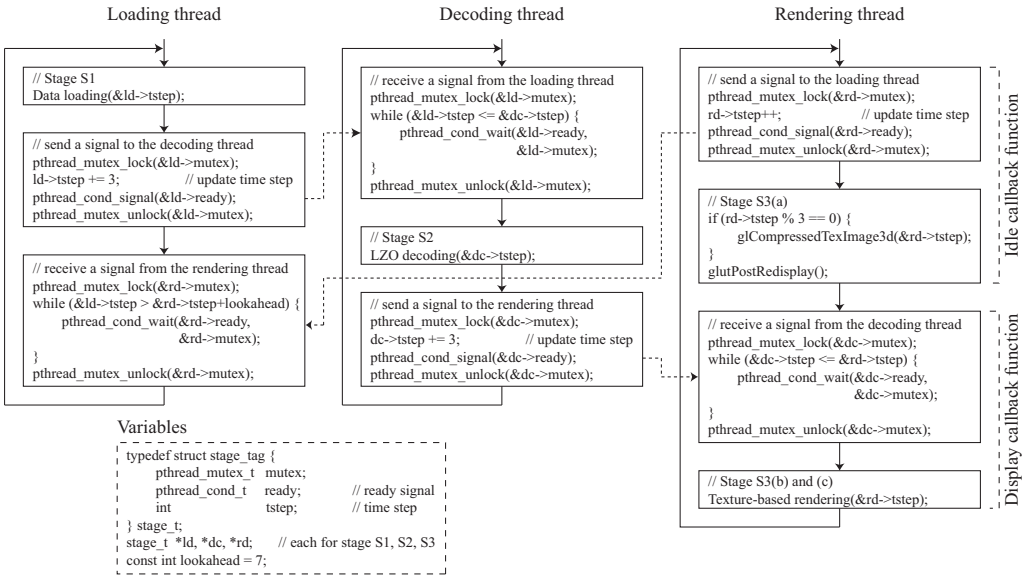


Fig. 5. Pseudocode of our thread-based pipeline mechanism. See text for details. Variable "lookahead" limits the number of stream data in the pipeline. This pipeline is allowed to process nine time steps of the data simultaneously, because each of the three stages can have the data containing three time steps.

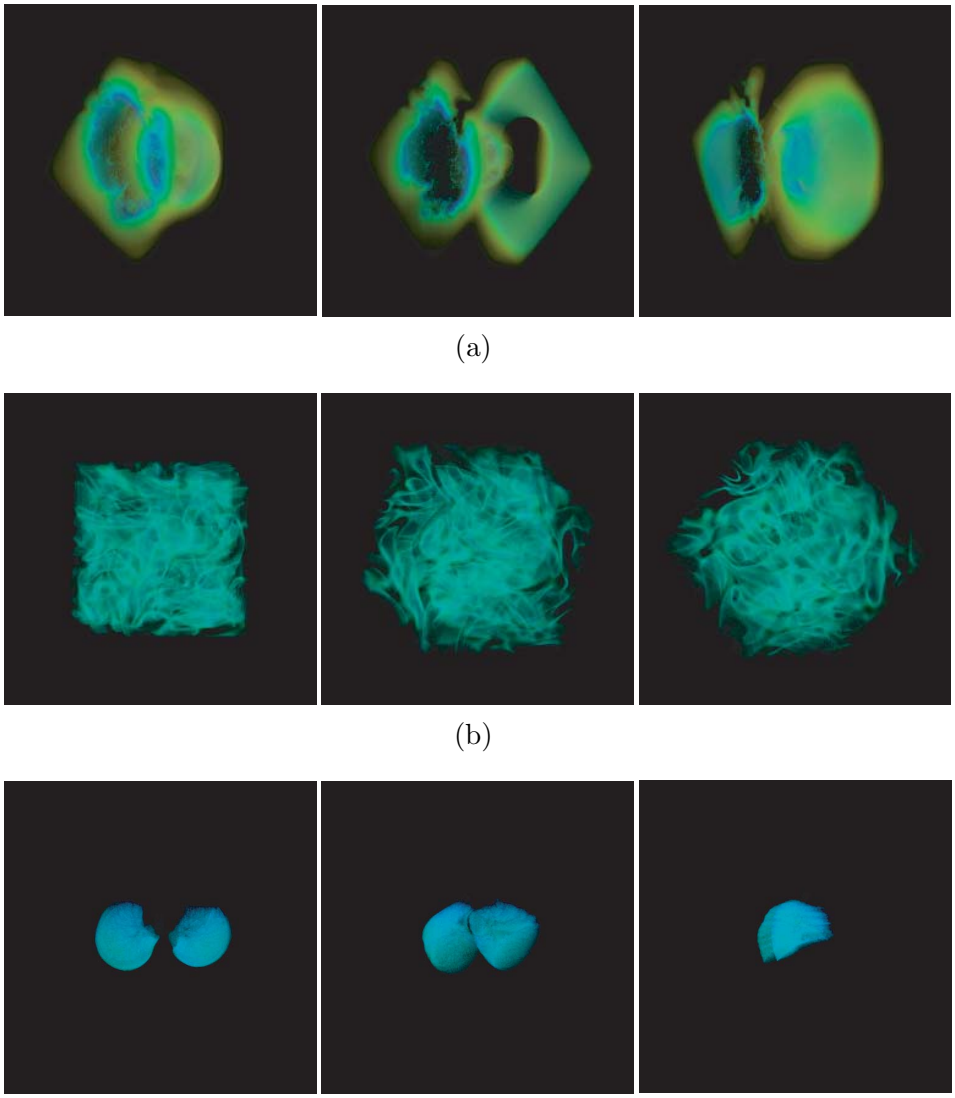


Fig. 6. Rendering results obtained from (a) D1: small jet, (b) D2: small vortex, and (c) D3: middle lung.

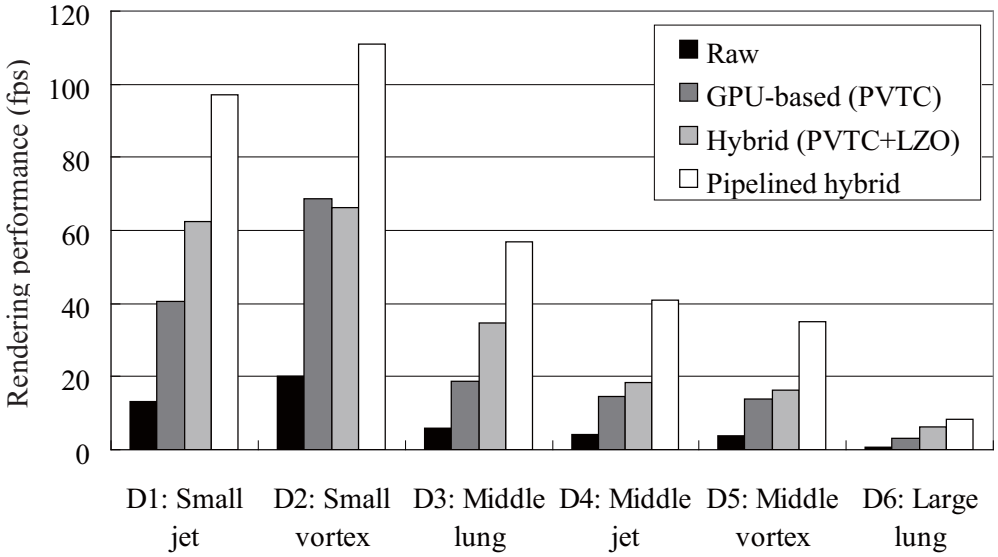


Fig. 7. Rendering performance in frames per second (fps). The performance is the average value.

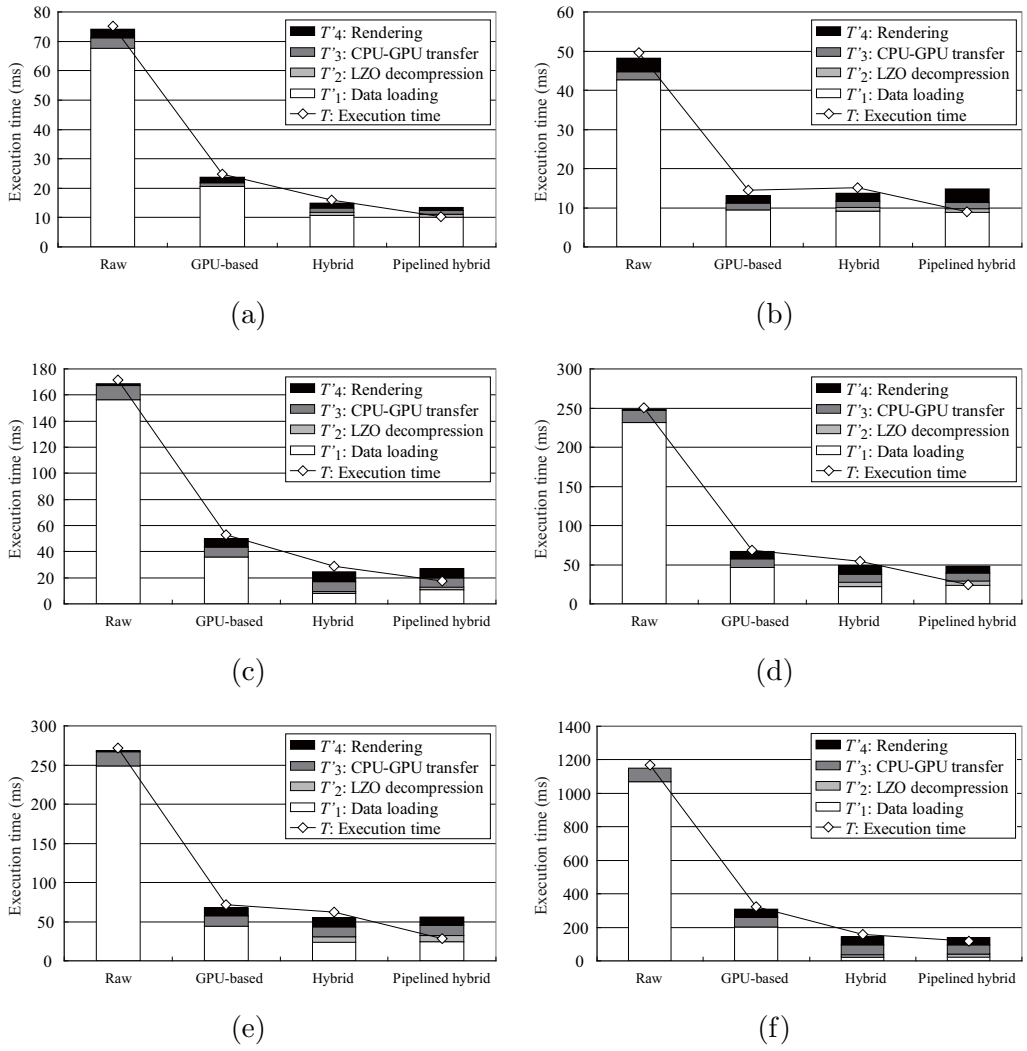


Fig. 8. Breakdown analysis of execution time required for rendering of a volume. Each of subfigures (a)–(f) corresponds to datasets D1–D6, respectively, showing the average time spent for rendering of a single time-step data.

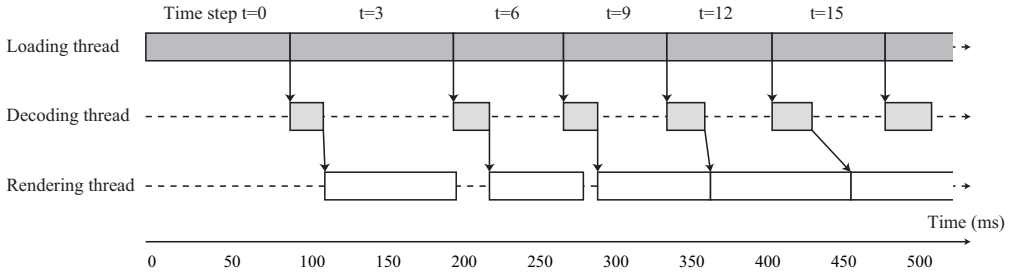


Fig. 9. Timeline chart showing how dataset D5 is rendered by the pipelined hybrid method. A rectangle represents a busy state of the corresponding thread.

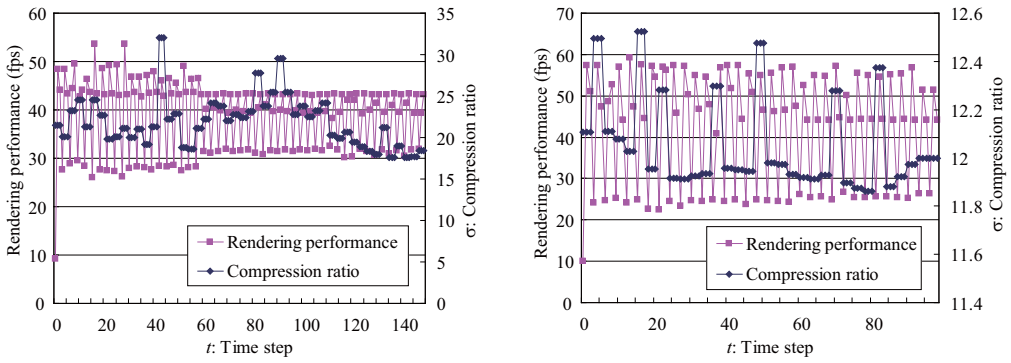


Fig. 10. Compression ratio and rendering performance for (a) datasets D4 and (b) D5 with different time steps.

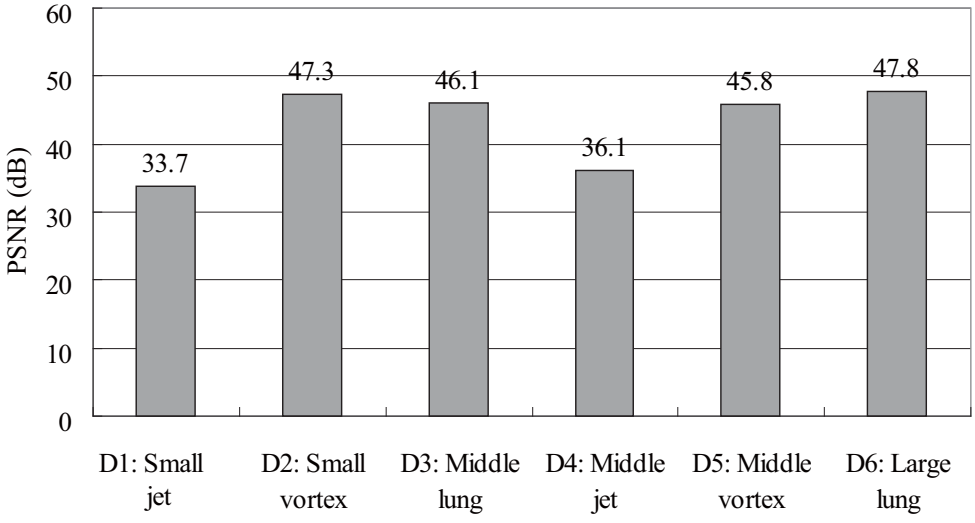


Fig. 11. Image quality of rendering results. PSNR values are averaged over all time steps.

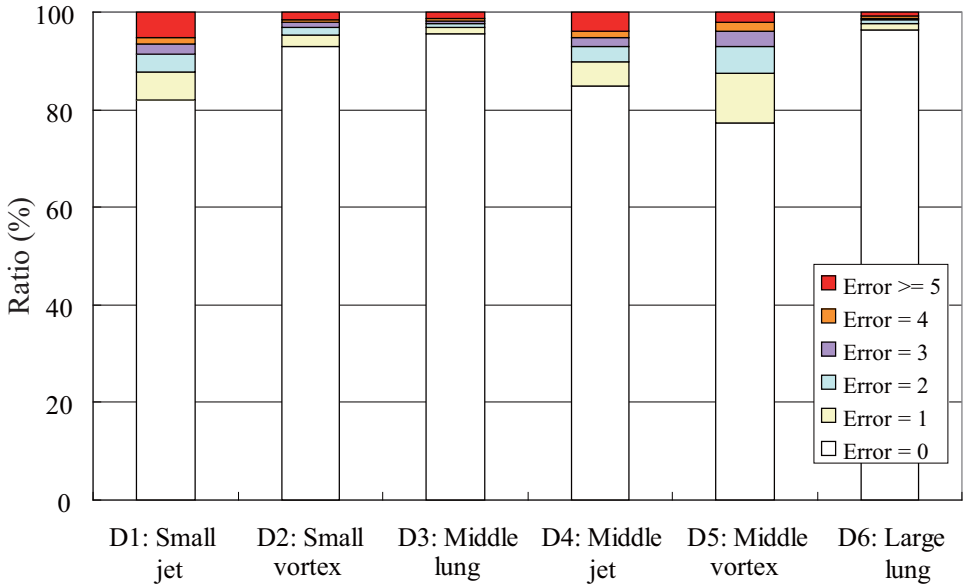


Fig. 12. Breakdowns of pixel errors in rendered images. Pixel values are in the range [0, 255].

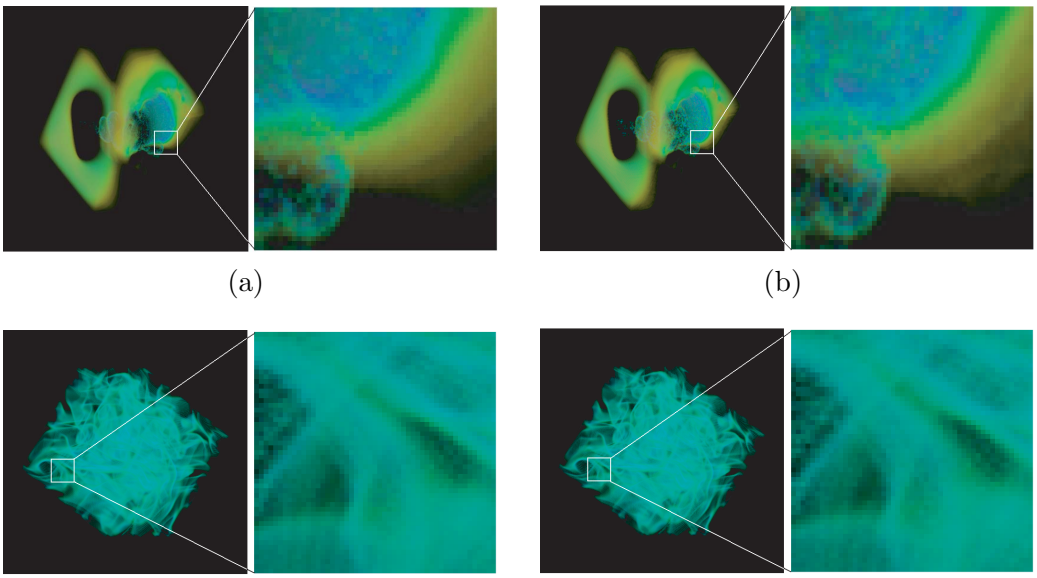
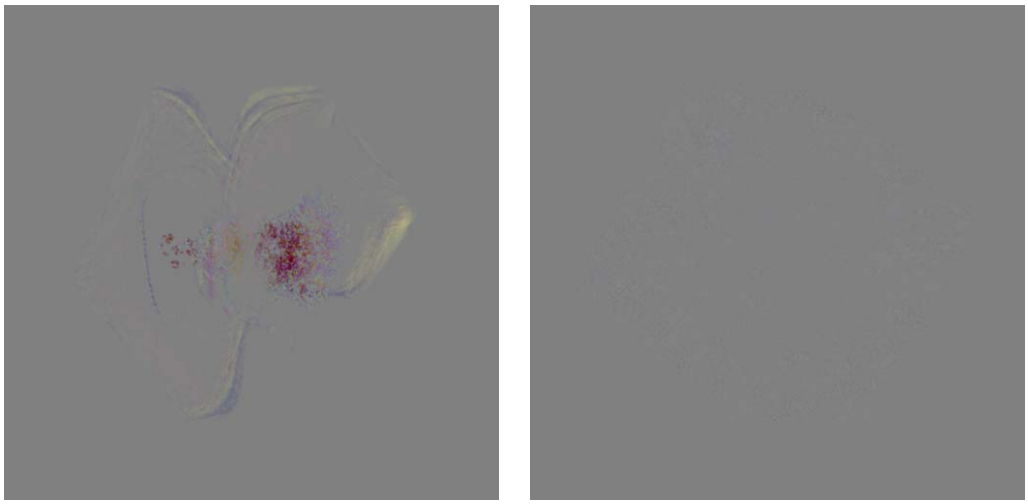


Fig. 13. Comparison of rendering results using datasets D4 at time step 20 and D5 at time step 65. The left-hand side is the result from raw data while the right-hand side is that from PVTC-compressed data.



(a) (b)

Fig. 14. Subtraction images for (a) datasets D4 and (b) D5, each produced from a pair of rendering results shown in Fig. 13. Subtracted pixels are rendered on the gray campus.

Table 1
Comparison of rendering systems. Compression ratio σ is given by $\sigma = D_1/D_2$, where D_1 and D_2 represent raw size and reduced size, respectively. Note that this table does not show a fair comparison, because compression ratio and image quality are not measured under the same condition, such as datasets, machines, and transfer functions.

Strategy	System	Pipelined decompression	Lossless compression	Supported coherence	Ratio σ	Image quality (dB)
Hybrid	This paper	Yes	No	Temporal and spatial	6–72	PSNR=33–48
	[10]	No				
GPU	[3]	No	Yes	Spatial	1.5–50	—
	[4]	No	Yes	Temporal and spatial	7–154	—
	[5]	No	No	Temporal or spatial	3.2–3.4	PSNR=32–71
	[6]	No	No	Temporal and spatial	8–60	SNR=10–26
	[7]	No	No	Temporal	2–8	PSNR=28–49
CPU	[8]	No	Yes	Spatial	—	—
	[9]	Yes	No	Temporal	7.1–34.7	SNR=5–43

Table 2
Datasets used for experiments.

Dataset	Volume size (voxel)	Time step	Raw file size per time step (MB)	Compression ratio	
				PVTC	PVTC+LZO
D1: small jet	$129 \times 129 \times 104$	150	1.7	6	11.5
D2: small vortex	$128 \times 128 \times 128$	99	2.0		6.5
D3: middle lung	$256 \times 256 \times 148$	411	9.3		67.0
D4: middle jet	$258 \times 258 \times 208$	150	13.2		21.5
D5: middle vortex	$256 \times 256 \times 256$	99	16.0		12.0
D6: large lung	$512 \times 512 \times 295$	411	73.8		71.7