Two-Stage Compression for Fast Volume Rendering of Time-Varying Scalar Data

Daisuke Nagayasu* Osaka University
Fumihiko Ino† Osaka University
Kenichi Hagihara‡ Osaka University

Abstract

This paper presents a two-stage compression method for accelerating GPU-based volume rendering of time-varying scalar data. Our method aims at reducing transfer time by compressing not only the data transferred from disk to main memory but also that from main memory to video memory. In order to achieve this reduction, the proposed method uses packed volume texture compression (PVTC) and Lempel-Ziv-Oberhumer (LZO) compression as a lossy compression method on the GPU and a lossless compression method on the CPU, respectively. This combination realizes efficient compression exploiting both temporal and spatial coherence in time-varying data. We also present experimental results using scientific and medical datasets. In the best case, our method produces 56% more frames per second, as compared with a single-stage (GPU-based) compression method. With regard to the quality of images, we obtain permissible results ranging from approximately 30 to 50 dB in terms of PSNR (peak signal-to-noise ratio).

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.3 [Computer Graphics]: Pixture/Image Generation—Display algorithms; E.4 [Coding and Information Theory]: Data Compaction and Compression;

Keywords: large-scale data visualization, data compression, temporal coherence, spatial coherence, graphics hardware

1 Introduction

Volume rendering (VR) [Akenine-Möller and Haines 2002] plays an increasingly important role in intuitive understanding of complex phenomena in various fields such as fluid dynamics, life sciences, and so on. In these fields, data becomes enormous with the rapid advance of computing systems. For example, recent X-ray computed tomography (CT) scans are capable of producing time-varying three-dimensional (3-D) data instead of a conventional 3-D snapshot. Thus, fast rendering of time-varying volume data is required to assist scientists in time series analysis [Ma 2003b].

One challenging issue in time-varying volume visualization is the data size, which usually exceeds memory capacity. For example, a volume of $512 \times 512 \times 512$ voxels with 100 time steps requires 12.5 GB of memory if each voxel has 1-byte data. Therefore, it is not easy to store the entire data in main memory and apparently in video memory. To solve this issue, we need out-of-core algorithms, which store the data in disk and dynamically transfer a portion of the data to video memory in an ondemand fashion.

There are two technical issues to be resolved for fast out-of-core VR:

I1. Fast data transfer from storage devices to video memory;

I2. Fast VR of 3-D data.

To address the above issues, prior methods [Akiba et al. 2005; Fout et al. 2005; Lum et al. 2002; Schneider and Westermann 2003; Binotto et al. 2003; Strengert et al. 2005] usually employ (1) data compression techniques to minimize transfer time and (2) hardware-acceleration techniques to minimize rendering time. Most methods typically transfer pre-compressed data from storage devices to video memory, and then perform volume projection using the graphics processing unit (GPU) [Fernando 2004; Montrym and Moreton 2005] or parallel computers [Strengert et al. 2005; Gao et al. 2005; Takeuchi et al. 2003].

Since data can be decompressed using the CPU and/or GPU, there are three possible strategies for the decompression problem. Suppose here that we have a GPU-based strategy that uses only the GPU for data decompression. The main drawback of this strategy is that it is not easy to achieve both fast decompression and high compression ratio due to the limitation of the GPU architecture. For example, current GPUs do not have large video memory (currently, at most 1 GB). Furthermore, their stream architecture [Khailany et al. 2001] usually slows down the processing speed against complex, irregular computation including branching statements.

In contrast, suppose also that we have a CPU-based decompression strategy that uses only the CPU instead of the GPU. This strategy will also suffer from low performance, because raw data is transferred from main memory to video memory. Therefore, both the GPU and CPU should be used for data decompression to address issue I1, as proposed in [Akiba et al. 2005].

In this paper, we present a two-stage compression method, aiming at achieving fast VR of time-varying scalar data. Our method addresses issue I1 by combining two compression/decompression methods M_g and M_c, each running at the GPU side and at the CPU side, respectively. Similar to Akiba's method [Akiba et al. 2005], the proposed method transfers compressed data through the entire data path from storage devices to video memory. With regard to issue I2, we employ a 3-D texture-based VR method [Cabral et al. 1994; Hadwiger et al. 2002] that makes use of hardware components in the GPU.

The proposed method differs from Akiba's method in using a different combination of methods M_g and M_c, which is designed to enable better cooperation between the GPU and CPU. For M_g, we use an extended version of volume texture compression (VTC) [OpenGL Extension Registry 2004b], which exploits hardware acceleration on the GPU. For M_c, on the other hand, we use Lempel-Ziv-Oberhumer (LZO) compression [Oberhumer 2005], taking advantage of larger CPU caches (as compared to GPU caches) for real-time decompression.

*d-nagayasu@ist.osaka-u.ac.jp
†ino@ist.osaka-u.ac.jp
‡hagihara@ist.osaka-u.ac.jp

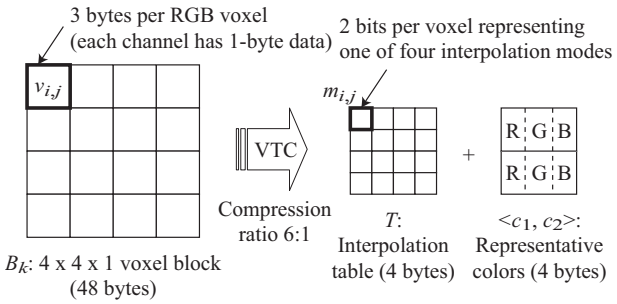


Figure 1: Overview of volume texture compression (VTC).

Our extended version of VTC, named packed VTC (PVTC), has two variations depending on the data structure: (1) PVTC for temporal encoding (PVTC-TE) and (2) PVTC for spatial encoding (PVTC-SE). These variations focus on the following coherences in time-varying data.

Temporal coherence. The correlation between voxel values observed at different moments in time.

Spatial coherence. The correlation between voxel values at different voxels in space.

The rest of the paper is organized as follows. We begin in Section 2 by introducing VTC and LZO, namely the basis of our method. We then present our two-stage compression method in Section 3 and show some experimental results in Section 4. Section 5 introduces related work. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 Volume Texture Compression (VTC)

VTC [OpenGL Extension Registry 2004b] is a lossy compression method for volume data that consists of voxels having color (RGB) and opacity (A) channels. It extends S3 texture compression (S3TC) [Iourcha et al. 1999; OpenGL Extension Registry 2004a] to support 3-D textures instead of 2-D textures. VTC is standardized as a part of OpenGL extensions [Shreiner et al. 2003] that are available on nVIDIA GPUs [Montrym and Moreton 2005]. There are four variations of the VTC algorithm in terms of the internal format of the input data. Among them, we use COMPRESSED_RGB_S3TC_DXT1_EXT, which provides the highest compression ratio for data in the RGB format.

Figure 1 shows an overview of VTC. It 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 a compressed texture. This algorithm replaces each of the blocks, namely 16 voxel values $v_{1,1}, v_{1,2}, \cdots, v_{4,4}$, with a sequence of data containing (1) two representative values $\langle c_1, c_2 \rangle$ and (2) a table $T = \{m_{i,j} \mid 1 \le i \le 4, 1 \le j \le 4\}$, where $m_{i,j}$ represents the linear interpolation mode for voxel $v_{i,j}$. In summary, VTC is a lossy compression method that approximates voxel values by linear interpolation of two representative values. This interpolation is independently done for each block, and thus VTC exploits spatial coherence in a small block. As shown in Figure 1, the compression ratio is 6:1, because VTC replaces each 48-byte block with 8-byte data.

Compressed textures can be decompressed by computing interpolated values for each voxel. For example, if we have $m_{i,j} = 3$ in table T, then the interpolated value for voxel $v_{i,j}$ is given by $(c_1 + 2c_2)/3$, according to the predefined scheme [OpenGL Extension Registry 2004a]. Note here that on-the-fly decompression is provided by VTC. Therefore, the entire volume is not decompressed at a time, avoiding additional (five times more) memory usage for decompressed data. Furthermore, this on-the-fly decompression is fast because it is implemented as a hardware component in the GPU. Since this component automatically recognizes whether textures are compressed or not, no program modification is needed to decode compressed textures.

2.2 Lempel-Ziv-Oberhumer (LZO) Compression

LZO [Oberhumer 2005] is a lossless, real-time data compression/decompression library based on LZ77 coding [Ziv and Lempel 1977]. Similar to VTC, LZO has various compression modes. In this paper, LZO denotes LZO1X-1, which is considered the fastest mode in most cases.

LZ77 is a dictionary coder, which scans the input data to replace repeating data with references back to the original data. It has a buffer, called a sliding window, to keep recent scanned data. If a longest match is found in the sliding window, it performs the replacement mentioned above.

Therefore, LZO will obtain high compression ratio if the same data sequence frequently appears in the sliding window. Furthermore, LZO 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.

Note here that the window size in the LZO library is 64 KB, and thus the sliding window can be stored entirely in CPU caches. This size is equivalent to 8192 VTC compressed blocks, which correspond to $512 \times 256 \times 1$ RGB voxels.

3 Two-Stage Data Compression

In this section, we first show requirements for methods M_c and M_g. We then describe the design of the proposed method, explaining how PVTC-TE and PVTC-SE satisfy the requirements.

3.1 Requirements for Efficient Compression

When combining two different compression methods M_c and M_g (see Figure 2), these methods must satisfy three requirements to achieve fast VR of time-varying data. In the following discussion, let T_1 and T_2 be the execution time for data transfer from disk to main memory and that for data decompression, respectively.

The first requirement is a constraint to each of the compression methods. Compression methods generally have to satisfy the following requirement in order to obtain timing reduction effects.

R1. Both M_c and M_g achieve high compression ratio sufficient to pay decompression cost T_2.

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 transfer time T_1, it is not a satisfactory one if it fails to reduce the total time $T_1 + T_2$.

On the other hand, the second and third requirements are inherent in two-stage compression methods.

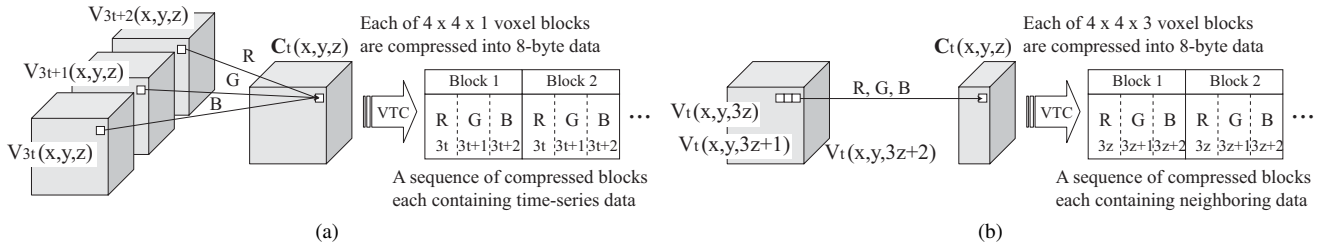


Figure 3: Data compression using PVTC. (a) Time-series scalar voxels in the same location are packed into an RGB voxel in PVTC-TE while (b) neighboring scalar voxels at the same time step are packed into an RGB voxel in PVTC-SE. This data packing generates a sequence of compressed blocks, each containing time-series data or neighboring data.

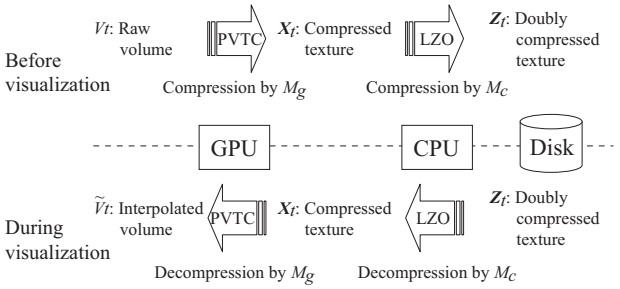


Figure 2: Overview of our two-stage compression method. Our method is a lossy compression method, because PVTC approximates the raw data with interpolated data.

R2. M_c and M_g exploit different coherences.

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.

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

This requirement 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.

3.2 Design Aspects

To satisfy requirements R1, R2, and R3, our method uses PVTC as M_g and LZO [Oberhumer 2005] as M_c. Figure 2 shows an overview of the proposed method. In the following discussion, let V_t denote the volume at time step t, where $t \geq 0$. Let $V_t(x,y,z)$ denote the voxel value of point (x,y,z) in volume V_t.

PVTC is a lossy compression method for time-varying volume data containing scalar values. The novelty of PVTC is the data packing that exploits all RGB channels of the inputs given to VTC [OpenGL Extension Registry 2004b]. This data packing here intends to exploit temporal or spatial coherence in time-varying volume data, which is defined over the time domain and the space domain. Thus, we propose two variations of PVTC, named PVTC-TE and PVTC-SE, which exploit temporal and spatial coherence, respectively. The details of each variation are as follows (see also Figure 3).

PVTC-TE. This variation packs three scalar voxels $V_{3t}(x,y,z)$, $V_{3t+1}(x,y,z)$, and $V_{3t+2}(x,y,z)$, namely time-series voxels in the same location, into an RGB voxel.

PVTC-SE. It packs three scalar voxels $V_t(x,y,3z)$, $V_t(x,y,3z+1)$, and $V_t(x,y,3z+2)$, namely neighboring voxels at the same time step, into an RGB voxel.

From the viewpoint of the requirements stated in Section 3.1, PVTC and LZO will satisfy requirement R1, because they provide fast decompression as follows.

- VTC compressed textures are decompressed by hardware components in the GPU. Furthermore, the GPU exploits two parallelisms to further accelerate this decompression: (1) single instruction multiple data (SIMD) instructions [Grama et al. 2003] and (2) vector instructions, allowing us to simultaneously process different voxels in a volume and different channels in a voxel, respectively.
- LZO is a real-time data compression/decompression library based on LZ77, which offers low-cost decompression as we mentioned in Section 2.2.

With respect to requirement R2, the combination of PVTC and LZO satisfies this requirement, because PVTC and LZO focus on different coherences as follows.

- PVTC-TE exploits temporal and spatial coherence in a small block while PVTC-SE exploits spatial coherence in a small block.
- In contrast, LZO exploits spatial coherence across small blocks, because it receives PVTC compressed textures as inputs.

This can be further explained as follows. The sliding window in LZO is equivalent to 8192 blocks, as we mentioned in Section 2.2. Given such blocks, LZO tries to find the longest match in these blocks. Therefore, LZO is responsible for exploiting coherence across small blocks. If we combine PVTC-SE with LZO, these 8192 blocks contain a single time step of $512 \times 256 \times 3$ scalar voxels, because three neighboring scalar voxels are packed into an RGB voxel. On the other hand, PVTC-SE performs interpolation against a block of $4 \times 4 \times 3$ scalar voxels. Thus, both LZO and PVTC-SE exploit spatial coherence in the volume but with different responsible areas. The coherence in small $4 \times 4 \times 3$ area is due to PVTC-SE while that in large $512 \times 256 \times 3$ area is due to LZO.

Finally, requirement R3 can be satisfied as follows.

- PVTC can keep the coherence between different blocks in the input volume, because it will encode similar blocks into com-

```
// The idle callback function registered by glutIdleFunc()
void Loading()
{
    t = (t < t_max) ? t++ : 0; // Update time step t
    if (t%3 == 0) {
        Step 1; // See text for details
        Step 2;
        glCompressedTexImage3d(compressed texture X_t); // Step 3
    }
    glutPostRedisplay(); // Call the display callback function
}
```

```
// The display callback function registered by glutDisplayFunc()
void Rendering()
{
    if (t%3 == 0) {
        cgGLBindProgram(PVTC_TE_R.cg); // See Figure 4(b)
    } else if (t%3 == 1) {
        cgGLBindProgram(PVTC_TE_G.cg);
    } else {
        cgGLBindProgram(PVTC_TE_B.cg);
    }
    Update the angle of compressed texture X_t; // glRotatef()
    Slice X_t perpendicular to the viewing direction; // glVertex3f()
    glutSwapBuffers(); // Display the produced image
}
```

(a)

```
// A fragment program for obtaining R channel data in compressed texture X_t
void PVTC_TE_R(
    in float4 texCoord : TEXCOORD0, // Texture coordinates normalized in the range [0,1]
    out float4 color : COLOR0, // Output color
    uniform sampler3D PVTC_Texture : TEXUNIT0) // Compressed texture X_t
{
    color = tex3D(PVTC_Texture, texCoord).rrrr; // Specify gggg or bbbb if G or B channel data is required.
    color = Transfer_Function(color); // Step 6
}
```

(b)

Figure 4: Overview of PVTC-TE implementation with branch elimination. (a) The CPU program and (b) fragment program written using the OpenGL library and the Cg toolkit. Branching statements are moved from the fragment program to the CPU program. See text for details of steps 1–6.

pressed blocks with similar representatives and tables. Thus, the coherence between blocks in the raw volume is also kept in compressed textures for later compression by LZO.

- The coherence mentioned above is expressed in the format of compressed textures, 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 the GPU and CPU architectures as follows.

- The GPU has hardware acceleration capability which the CPU does not have. VTC is accelerated using this capability with SIMD and vector instructions.
- The CPU has larger caches as compared to texture caches in the GPU. Larger caches are suited to duplicating operations in LZO decompression, because these operations refer large area in the sliding window, which cannot be stored in small texture caches. In other words, the CPU cooperates with the GPU by providing larger caches to accelerate decompression for large area.

3.3 Data Compression

We now describe how the proposed method compresses the volume data. Let $\mathbf{C}_t$ denote the t-th input of VTC, namely the packed (raw) volume of RGB data, where $0 \leq t < t_{max}$. Let $\mathbf{C}_t(x,y,z)$ be the voxel value of point (x,y,z) in packed volume $\mathbf{C}_t$. Then, the compression procedure can be written as follows.

1. Data packing. The method packs the raw data into RGB data to obtain a packed volume $\mathbf{C}_t$, according to one of the following modes.
   - PVTC-TE mode. Three scalar values $V_{3t}(x,y,z)$, $V_{3t+1}(x,y,z)$, and $V_{3t+2}(x,y,z)$ are copied to R, G, and B channels of $\mathbf{C}_t(x,y,z)$, respectively. Empty values are added as padding data if necessary ($t_{max} \bmod 3 \neq 0$).
   - PVTC-SE mode. Three scalar values $V_t(x,y,3z)$, $V_t(x,y,3z+1)$, and $V_t(x,y,3z+2)$ are copied to R, G, and B channels of $\mathbf{C}_t(x,y,z)$, respectively. Empty values are also added if necessary ($Z \bmod 3 \neq 0$, where Z represents the volume size in z-axis direction).
2. Data compression by VTC. The packed volume $\mathbf{C}_t$ is given to VTC to obtain a compressed texture $\mathbf{X}_t$.
3. Data compression by LZO. The compressed texture $\mathbf{X}_t$ is given to LZO to obtain a doubly compressed texture $\mathbf{Z}_t$.

The above procedure is repeated with incrementing time step t.

3.4 Data Decompression

The proposed method repeatedly decompresses doubly compressed textures during visualization. Note here that PVTC-TE is allowed to load data at every three time steps, because each of compressed textures in PVTC-TE contains time-series data. The decompression is done in the following steps (see also Figures 4 and 5).

1. Data loading from storage devices. A doubly compressed texture $\mathbf{Z}_{\lfloor t/3 \rfloor}$ ($\mathbf{Z}_t$, for PVTC-SE) is transferred from disk to main memory.
2. Data decompression by LZO. The CPU decompresses $\mathbf{Z}_{\lfloor t/3 \rfloor}$ ($\mathbf{Z}_t$, for PVTC-SE) to obtain a compressed texture $\mathbf{X}_{\lfloor t/3 \rfloor}$ ($\mathbf{X}_t$, for PVTC-SE).

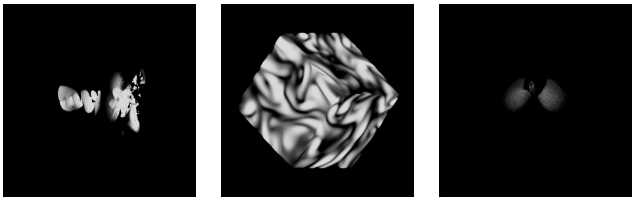
```
// A fragment program for obtaining the appropriate data in compressed texture X_t
void PVTC_SE(
    in float4 texCoord : TEXCOORD0, // Texture coordinates normalized in the range [0,1] for x- and y-coordinate
                                    // z-coordinate is specified in the range [0,Z − 1], where Z represents the volume size in z-axis direction
    out float4 color : COLOR0, // Output color
    uniform sampler3D PVTC_Texture : TEXUNIT0, // Compressed texture X_t
    uniform float zsize) // Texture size ⌈Z/3⌉ − 1 of X_t in z-axis direction
{
    float zCoord = floor(texCoord.z); // Omit fractions
    float zMod = fmod(zCoord, 3);
    zCoord = (zCoord−zMod)/3; // z-coordinate in X_t (in the range [0, ⌊Z/3⌋])
    // Data fetch from X_t using z-coordinate normalized in the range [0,1]
    color = tex3D(PVTC_Texture, float4(texCoord.x, texCoord.y, zCoord/zsize, texCoord.w)).rgba;
    // Select the appropriate channel
    if (zMod == 0) {
        color = color.rrrr;
    } else if (zMod == 1) {
        color = color.gggg;
    } else {
        color = color.bbbb;
    }
    color = Transfer_Function(color); // Step 6
}
```

Figure 5: Overview of PVTC-SE implementation without branch elimination. The CPU program is omitted due to the space limitation. It can be implemented similarly to PVTC-TE.

3. Data transfer from the CPU to the GPU. $\mathbf{X}_{\lfloor t/3 \rfloor}$ ($\mathbf{X}_t$, for PVTC-SE) is transferred from main memory to video memory.
4. VR of compressed textures. Texture-based VR [Cabral et al. 1994; Hadwiger et al. 2002] is carried out with the following steps 5 and 6.
5. Data decompression by PVTC. For all points (x,y,z) in V_t, linear interpolated value $\tilde{V}_t(x,y,z)$ is computed automatically by hardware components, as we mentioned in Section 2.1. That is, the GPU obtains $\tilde{V}_t(x,y,z)$ by simply accessing the corresponding packed volume $\mathbf{C}_t(x,y,z)$, according to one of the following modes.
    - PVTC-TE mode. The GPU refers R, G, and B channel data of $\mathbf{C}_{\lfloor t/3 \rfloor}(x,y,z)$ if t mod 3 = 0, 1, and 2, respectively.
    - PVTC-SE mode. The GPU refers R, G, and B channel data of $\mathbf{C}_t(x,y,\lfloor z/3 \rfloor)$ if z mod 3 = 0, 1, and 2, respectively.
6. Data classification by a transfer function. The final color and opacity of $\tilde{V}_t(x,y,z)$ are determined by a transfer function.

The first three steps 1–3 mentioned above are processed at the CPU side while the remaining steps are done at the GPU side. In our method, these two groups of steps are structured in a pipeline, allowing the CPU to load the next data just after pushing the current data to the GPU. Thus, our method overlaps CPU computation with GPU computation to obtain higher throughput from the pipeline. Note here that the throughput is limited by the bottleneck stage in the pipeline: steps 1–3 or steps 4–6.

Figure 4 shows an overview of our PVTC-TE implementation written using the OpenGL library [Shreiner et al. 2003] and the C for graphics (Cg) toolkit [Mark et al. 2003]. In this implementation, conditional branches on time step t are eliminated from the fragment program [Montrym and Moreton 2005] in order to obtain higher performance on the GPU. This elimination is done by moving branching statements to the CPU program and by selecting the appropriate fragment program according to time step t. Accordingly, we write three different fragment programs, each refers R, G, or B channel data in a compressed texture.



(a) (b) (c)

Figure 6: Produced images using (a) turbulent jet, (b) turbulent vortex, and (c) lung datasets.

In contrast, as shown in Figure 5, it is not easy for PVTC-SE to realize efficient branch elimination. This is due to the branch condition on z-coordinate. This condition is not suited to the GPU, which is designed to achieve higher performance by applying SIMD instructions to every point in the working texture. Therefore, to take advantage of SIMD instructions, namely to move branching statements from the fragment program to the CPU program, the raw volume must be decomposed into three parts (z mod 3 = 0, 1, and 2) in advance of data compression. We currently avoid this decomposition, because it increases runtime overhead due to volume restoration incurred during data decompression. Thus, our PVTC-SE implementation has branching statements in the fragment program, as shown in Figure 5.

4 Experimental Results

We have evaluated our method in terms of rendering performance, compression ratio, and image quality. The method is implemented using the C++ language, the OpenGL library [Shreiner et al. 2003],

Table 1: Datasets used for experiments.

Dataset	Volume size (voxel)	Time step	File size (MB)	Coherence	
				Temporal	Spatial
D1	$129 \times 129 \times 104$	99	163	Low	High
D2	$128 \times 128 \times 128$	99	198	Low	Low
D3	$256 \times 256 \times 148$	411	3802	High	High
D4	$258 \times 258 \times 208$	99	1307	Low	High
D5	$256 \times 256 \times 256$	99	1584	Low	Low

and the Cg toolkit [Mark et al. 2003]. We also have implemented another variation that employs zlib [Gailly and Adler 2005], a family of LZ77 algorithms, as method M_c. As compared to LZO, zlib usually provides a higher compression ratio but takes longer time for data decompression.

For experiments, we use a commodity PC equipped with 2 GB of main memory and 150 GB of serial ATA disk storage. The PC has a Pentium 4 CPU running at 3 GHz clock speed and an nVIDIA GeForce 6800 GTO card [Montrym and Moreton 2005] with 256 MB of video memory. The graphics card is connected to a PCI Express bus.

Table 1 summarizes the five datasets D1–D5 [Ma 2003a] used for experiments. See also Figure 6 for visualization results obtained from datasets D1–D3. The remaining datasets D4 and D5 have almost the same results as D1 and D2, because D4 and D5 are high-resolution versions of D1 and D2, respectively. Each dataset consists of voxels of 1-byte data and has the following characteristics in terms of coherence.

- D1. This dataset shows turbulent jets flowing from a nozzle. It has low temporal coherence due to the moving jets. However, it has high spatial coherence due to many transparent voxels.
- D2. Turbulent vortex flow is captured in this dataset. Both temporal and spatial coherence is low, because the vortices always move around the field and have few transparent voxels.
- D3. It shows a sequence of deformations of the lung. This sequence represents the process of nonrigid registration [Ino et al. 2005], which performs alignments between two 3-D images. Since the deformations are small, it has high temporal coherence. Furthermore, it has high spatial coherence because many voxels have similar values.

4.1 Rendering Performance

To demonstrate the performance gain of our method, we compare the method with three variations: (1) a straightforward method that transfers raw data from disk to video memory; (2) a single-stage method that uses PVTC on the GPU but no compression method on the CPU; and (3) a zlib-based two-stage method that combines PVTC with zlib instead of LZO.

In experiments, datasets are rendered on the screen with the appropriate size: a 256×256 pixel screen for D1 and D2 and a 512×512 pixel screen for the remaining datasets. The viewing direction is initially set to z-axis direction. It is then rotated 2 degrees around x- and y-axes when time step t is updated.

Figure 7 shows frame rates measured using each of the methods. Table 2 also shows the execution time T and its breakdown required for rendering of a single frame. The breakdown here consists of time T_1 and T_2, each representing the transfer time required to send data from disk to main memory and the decompression time spent by method M_c (LZO or zlib), respectively. Thus, time T_1 and T_2 correspond to step 1 and step 2 presented in Section 3.4. Further breakdown analysis is not presented because we could not measure the execution time of the remaining steps, which are processed by underlying graphics APIs.

Firstly, we analyze the performance of PVTC-TE. Figure 7(a) shows that the combination of PVTC-TE and LZO achieves higher performance, as compared with the straightforward method. Thus, the proposed method achieves a speedup ranging from a factor of 2.1 to 5.2 over the straightforward method. This performance gain is due to the reduction of the transfer time T_1, as shown in Table 2. Actually, we can see that this reduction contributes to the reduction of the total time T.

As compared with the single-stage (PVTC) method, the combination of PVTC-TE and LZO provides approximately 1–56% higher performance to all datasets except D2. In contrast, the performance is decreased by 10% for D2. This performance decrease indicates that the data size reduction provided by LZO was not enough to save time. We show this later in Section 4.2. Thus, LZO could not satisfy requirement R1 for D2, which has low temporal and spatial coherence.

By comparing LZO with zlib in two-stage methods, LZO achieves 1–6% higher performance for all datasets except D1. The breakdown analysis in Table 2 explains this performance gain over zlib. We can see that zlib has relatively shorter transfer time T_1 but requires two times longer time T_2 for data decompression. Due to this long time T_2, zlib fails to achieve shorter total time $T_1 + T_2$, as compared to LZO. However, with respect to small dataset D1, there is no significant difference between LZO and zlib. Thus, zlib might be useful to small datasets. However, we think that this performance gain is ignorable because data size is small.

Secondly, we analyze the performance of PVTC-SE presented in Figure 7(b). By comparing the single-stage method of PVTC-SE and that of PVTC-TE, we can see that PVTC-SE results in lower performance in all cases. Moreover, even if PVTC-SE is combined with LZO, this two-stage method shows almost the same performance ($\pm 2\%$) as the single-stage method. In particular, it fails to reduce the total time T for dataset D3, despite of significant reduction of time $T_1 + T_2$. Thus, the reduction of time $T_1 + T_2$ does not lead to the reduction of the total time T. This is caused by the branching statements in the fragment program. As we mentioned in Section 3.4, PVTC-SE requires conditional branches at step 5, which cannot be effectively processed on the GPU. Therefore, it takes relatively longer time at the GPU side, being the bottleneck stage in the pipeline (see Section 3.4). Thus, branching operations on the GPU limit the throughput of the pipeline. Therefore, even if time $T_1 + T_2$ is significantly reduced at the CPU side, this reduction will not lead to higher performance in PVTC-SE.

Note here that we have confirmed that the fragment program of PVTC-SE in Figure 5 achieves almost the same performance as that of PVTC-TE if branching statements are simply removed from the program. Although this removal apparently does not produce the correct results, it indicates that branchless fragment programs are necessary to achieve fast VR.

Finally, the transfer time T_1 in PVTC-SE is longer than that in PVTC-TE. This is due to the difference of the number of disk accesses required for a single time step. For example, PVTC-TE requires only a single data access to load data of three time steps, because it packs time-series data into a single compressed texture. In contrast, PVTC-SE requires three accesses to load three compressed textures. Therefore, PVTC-TE has relatively lower overhead for a single volume, achieving shorter transfer time T_1 for a single frame.

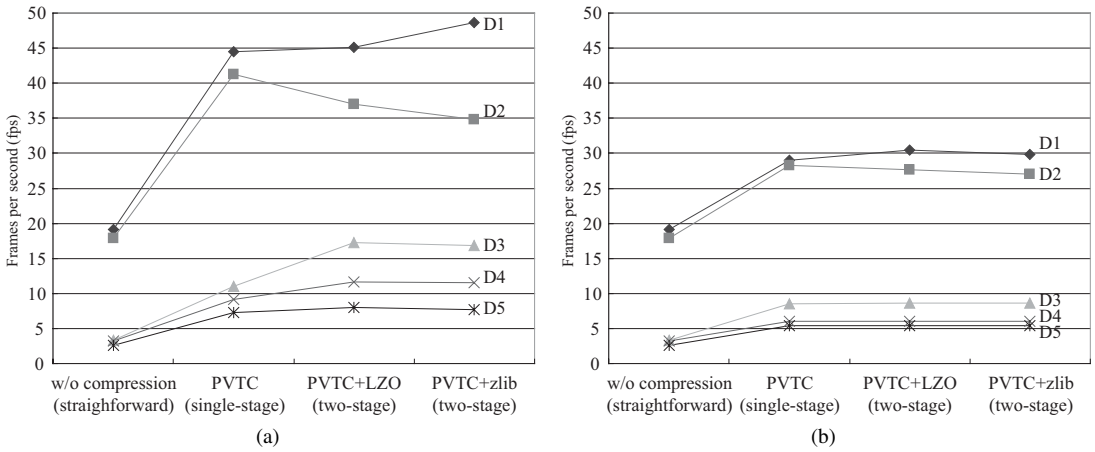


Figure 7: Rendering performance using (a) PVTC-TE and (b) PVTC-SE.

Table 2: Execution time required to render a frame (ms). T represents the total execution time required for rendering of a volume. T_1 and T_2 are the breakdown of time T, each representing the transfer time from disk to main memory and the decompression time spent by LZO or zlib, respectively.

Dataset	w/o compression		PVTC-TE								PVTC-SE							
			PVTC		PVTC+LZO			PVTC+zlib			PVTC		PVTC+LZO			PVTC+zlib		
	T		T		T			T			T		T			T		
	T_1		T_1		T_1	T_2		T_1	T_2		T_1		T_1	T_2		T_1	T_2	
D1	44	52	8.7	23	6.0	2.0	22	3.6	3.4	21	15.7	35	7.0	1.8	33	7.0	2.8	34
D2	49	56	9.7	24	10.4	2.4	27	8.5	5.5	29	15.2	35	14.1	2.5	36	14.3	5.4	37
D3	284	302	45.6	90	6.0	2.7	58	5.7	6.6	59	39.0	117	12.7	3.6	117	10.1	7.0	117
D4	285	315	51.5	109	16.4	9.7	86	9.7	16.8	87	52.3	167	23.6	8.0	165	20.1	14.8	165
D5	347	388	60.0	138	32.0	14.4	125	23.0	28.2	130	63.2	184	50.7	19.6	183	38.7	35.9	184

Table 3: Average compression ratio for datasets.

Dataset	PVTC-TE			PVTC-SE		
	PVTC	+LZO	+zlib	PVTC	+LZO	+zlib
D1	6.0	12.0	17.4	5.9	16.7	22.3
D2	6.0	6.5	8.0	5.9	6.3	7.7
D3	6.0	67.0	81.1	5.9	58.6	64.4
D4	6.0	22.5	39.4	6.0	28.1	42.2
D5	6.0	12.0	17.5	5.9	8.6	11.3

4.2 Compression Ratio

Table 3 shows the compression ratio for each dataset. While the single-stage (PVTC) method achieves a compression ratio of 6:1, the two-stage method contributes to obtain more compact data with reducing the data size to further 1/11–1/1.1. Note here that the compression ratio of the single-stage method is not always exactly 6:1. In some cases, it requires padding data, which decreases the compression ratio to approximately 5.9:1, as shown in Table 3.

The combination of PVTC-TE and LZO (or zlib) achieves a compression ratio of at least 12:1 for all datasets except D2. This means that LZO (or zlib) reduces the data size of PVTC compressed textures into at least half. Recall here that this combination fails to improve the performance for dataset D2 (see Section 4.1). Thus, in our experimental machine, the two-stage method is effective if the second-stage compression method further reduces the data size of compressed textures into half.

For datasets D1 and D4, PVTC-SE achieves higher compression ratio than PVTC-TE. For datasets D3 and D5, however, PVTC-SE fails to outperform PVTC-TE. These results are reasonable according to the design. That is, PVTC-SE, rather than PVTC-TE, is effective for data with low temporal coherence. On the other hand, PVTC-TE is effective for temporally coherent data.

For the remaining dataset D2, the second-stage compression method, namely LZO or zlib, is not so effective. The coherence in this dataset is not so high, so that the match found in the sliding window is not long enough to obtain high compression. Actually, the compression ratio is at most 1.1:1 even if we apply LZO or zlib directly to D2. Therefore, we think that lossy compression methods such as PVTC are required to achieve fast VR for low-coherence data.

Although PVTC is a lossy compression method, it allows us to render larger volume data on the GPU. For example, a GPU with 512 MB of video memory is not enough to store a raw volume of $1024 \times 1024 \times 1024$ voxels, which contains 1 GB data. In PVTC-SE, this data is reduced to 1/6, and thus approximately 170 MB of video memory is sufficient to store the entire volume. On the other hand, PVTC-TE must store a sequence of three volumes because it packs them into an RGB volume. Due to this data packing, it requires at least 512 MB (=1 GB/6*3) of video memory. Additional video memory is also required for the frame buffer.

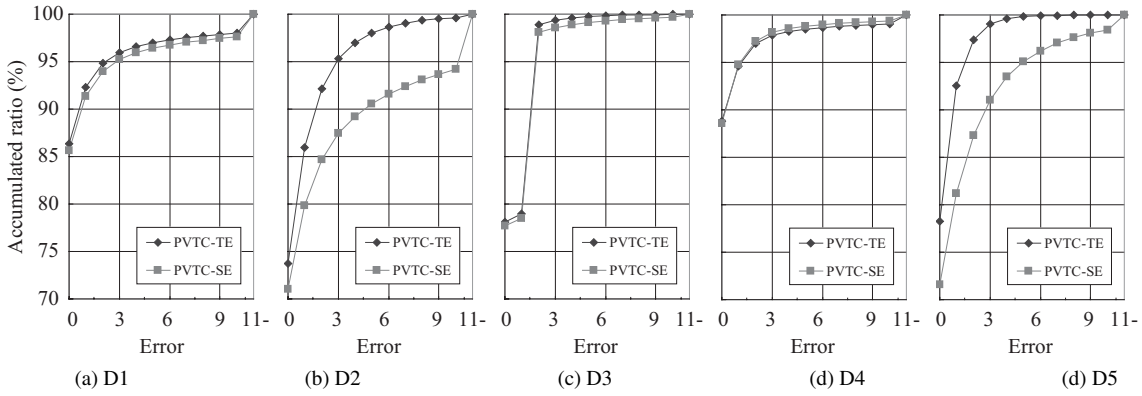


Figure 8: Accumulated ratio of errors in rendered images. Pixel values are in the range [0,255].

Table 4: Image quality of rendering results (in PSNR).

Dataset	PVTC-TE (dB)	PVTC-SE (dB)
D1	31.5	31.9
D2	43.5	28.4
D3	47.6	44.6
D4	35.7	39.9
D5	49.5	36.1

4.3 Image Quality

We measured PSNR (peak signal-to-noise ratio) to evaluate the quality of rendered images. Higher PSNR values represent higher qualities with fewer noises. In general, PSNR values of at least 30 dB are desired for rendered images, and it is hard to see image degradation if PSNR values are higher than 40 dB.

Table 4 shows PSNR values measured for each dataset. In most cases, we observe PSNR values of at least 30 dB. These results are competitive to prior work [Lum et al. 2002] that achieves similar qualities ranging from 28.4 to 48.9 dB. Because we observe higher PSNR values for high-coherence dataset D3, our method minimizes image degradation against high-coherence data.

Figure 8 shows the accumulated ratio of errors in produced images. The accumulated ratio of 80% at error 6 means that 80% of pixels have an error within pixel value 6, as compared to the correct image produced by the straightforward method.

In Figure 8(b), we can see that approximately 5% of pixels have an error of at least pixel value 10. This means that the interpolation in PVTC-SE was not so accurate in terms of image quality. Therefore, PVTC-SE degrades the image quality for dataset D2. Due to the same reason, we also observe image degradation for dataset D5. Thus, PVTC-TE provides higher quality images than PVTC-SE in these cases. However, there is no significant difference between PVTC-TE and PVTC-SE except the cases mentioned above. Both variations produce exact values for 80% of pixels.

Figure 9 presents a pair of images, each produced from the raw data and from the PVTC compressed data. By comparing them in a zoomed view, we can see some block artifacts in Figure 9(b), especially where neighboring pixels do not have similar values. Similar tendencies are observed also for other datasets. However, these artifacts do not significantly change the overall appearance of produced images. Thus, we think that the image quality of PVTC is permissible to assist users in time-series analysis.

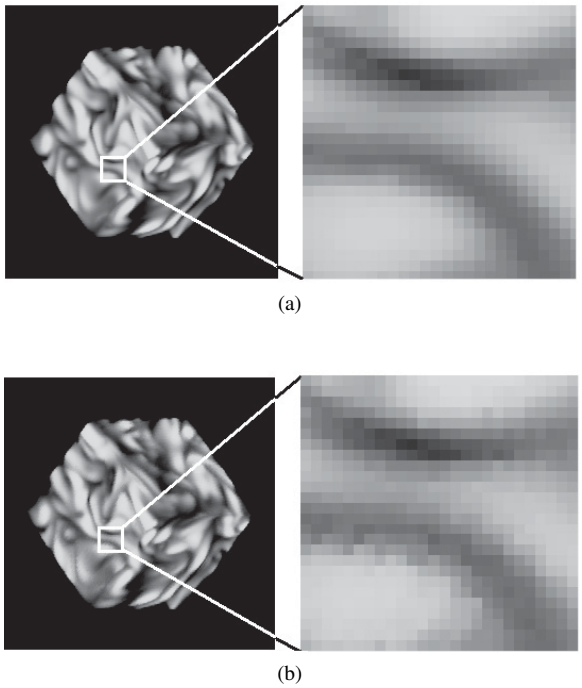


Figure 9: Comparison of rendering results (a) without compression and (b) with PVTC-TE. Both images are rendered for dataset D2 at the same time step.

5 Related Work

As we mentioned in Section 1, prior methods can be classified into three groups in terms of where data is decompressed: (1) the two-stage methods [Akiba et al. 2005], (2) the GPU-based methods [Lum et al. 2002; Schneider and Westermann 2003; Binotto et al. 2003; Fout et al. 2005], and (3) the CPU-based methods [Strengert et al. 2005]. Table 5 summarizes their differences.

To the best of our knowledge, only [Akiba et al. 2005] use both the CPU and GPU for data decompression. In their method, the CPU performs lossless compression [Clyne 2003] based on wavelet transformation, which allows the selection of an appropriate resolution level from the raw data. On the other hand, the GPU partitions

Table 5: Comparison of our method with prior compression-based methods.

Method	Data decompression		Lossless compression	Exploited coherence	
	CPU	GPU		Temporal	Spatial
PVTC-TE	Yes	Yes	No	Yes	Yes
PVTC-SE	Yes	Yes	No	No	Yes
[Akiba et al. 2005]	Yes	Yes	Yes*	No	Yes
[Fout et al. 2005]	No	Yes	No	No	Yes
[Lum et al. 2002; Fout et al. 2005]	No	Yes	No	Yes	No
[Schneider and Westermann 2003]	No	Yes	No	Yes	Yes
[Binotto et al. 2003]	No	Yes	Yes	Yes	Yes
[Strengert et al. 2005]	Yes	No	Yes	No	Yes

*: Voxel values outside the user-defined domain of interest are assigned transparent opacities.

the volume into 3-D blocks, which are then reduced by replacing the same blocks with references to the representative block. Thus, the GPU takes the responsibility for exploiting the coherence across 3-D blocks, as we do this by LZO on the CPU. The advantage of their method is the multiresolution representation that allows us to choose between interactivity and image quality at runtime. However, the compression ratio for the turbulent jet dataset is not so high (2:1), and thus it is not clear whether the CPU cooperates well with the GPU. On the other hand, our method is designed to provide better cooperation with exploiting the architectural advantages of the CPU and GPU. Furthermore, our method exploits temporal coherence, but with lossy compression.

On the other hand, many methods use only the GPU for data decompression. These GPU-based methods can be further classified into three groups in terms of exploited coherence.

- Spatial coherence. [Fout et al. 2005] pack $2 \times 2 \times 2$ neighboring voxels into a vector, and then apply quantization to the vector to approximate it with a representative vector. Thus, this method is a lossy compression method, like as PVTC. However, since this vector quantization method has more approximation modes than PVTC, it provides higher quality (32–78 dB) but with lower compression ratio (3:1). Vector quantization produces a sequence of data containing representative vectors, namely a codebook, and pointers to the representative vectors. This sequence is more irregular than PVTC compressed textures, which encode data without using arbitrary pointers. Therefore, a combination of vector quantization and LZO seem not be a better solution than our combination, because the match found by LZO will not be so long due to irregular inputs.

- Temporal coherence. In addition to the variation mentioned above, [Fout et al. 2005] also propose another variation that exploits temporal coherence. Like the relation between PVTC-TE and PVTC-SE, this variation packs time-series voxels instead of neighboring voxels.

  [Lum et al. 2002] also exploit temporal coherence by using the discrete cosine transform (DCT), which transforms 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 method allows us to choose the compression ratio to control image quality.

- Spatial and temporal coherence. [Schneider and Westermann 2003] present a compression method based on vector quantization. This method initially partitions the volume into disjoint blocks of size $4 \times 4 \times 4$, then recursively downsamples each block to create a hierarchical data structure composed of three vectors with length 64, 8, and 1, respectively. Each vector stores the difference between the original data and the respective down-sampled value. It exploits temporal coherence by performing vector quantization using the same codebook between successive time steps.

  [Binotto et al. 2003] realize a lossless compression method that partitions the volume into 3-D blocks, which are then reduced by replacing the same blocks with the representative block. This replacement is also done between different time steps, and thus temporal coherence is exploited with spatial coherence. Their method is effective for highly sparse and temporally coherent data.

Finally, a CPU-based method is proposed in [Strengert et al. 2005]. This method is based on [Guthe et al. 2002], exploiting spatial coherence at each time step. It employs wavelet transformation to generate a hierarchical octree, which is then compressed by an entropy encoding. Similar to other wavelet-based methods, this method provides multiresolution visualization. Their method might be useful if the GPU, rather than the CPU, is a bottleneck in the pipeline. However, the GPU is rapidly increasing its performance beyond Moore's low [Moore 1965; Montrym and Moreton 2005]. Therefore, we think that CPU-based methods will not be so effective on future environments, where the faster GPU will wait for data coming from the slower CPU. In such environments, the GPU should assist the CPU to stream more data through the entire data path from storage devices to video memory.

6 Conclusion

We have presented a two-stage compression method for accelerating GPU-based VR of time-varying scalar data. Our method combines PVTC with LZO, each running on the GPU and CPU, respectively. The key contribution of the method is PVTC, which aims at establishing better cooperation between the GPU and CPU. PVTC has two variations, namely PVTC-TE and PVTC-SE, which focus on temporal and spatial coherence, respectively. In both variations, PVTC takes the responsibility for exploiting the spatial coherence in a small block while LZO is responsible for exploiting the spatial coherence across blocks. From the viewpoint of architectural design, PVTC is accelerated by hardware components in the GPU with on-the-fly capability while LZO is accelerated by larger caches in the CPU with real-time design.

In experiments, the combination of PVTC-TE and LZO produces 56% more frames per second in the best case, as compared with the single-stage compression method that uses PVTC-TE on the GPU. Furthermore, this combination achieves a speedup ranging from a factor of 2.1 to 5.2 over the straightforward method that does not use compression methods. With respect to the quality of images,

we obtain permissible quality ranging from approximately 30 to 50 dB in terms of PSNR.

Our future work is to improve the scalability on the volume size, which is limited by the capacity of video memory. Although our out-of-core VR method provides a temporally scalable framework, the method will be more useful if it is also spatially scalable.

Acknowledgments

This work was partly supported by JSPS Grant-in-Aid for Scientific Research for Scientific Research (B)(2)(18300009) and on Priority Areas (17032007). We would like to thank the anonymous reviewers for their valuable comments.

References

AKENINE-MÖLLER, T., AND HAINES, E., Eds. 2002. *Real-Time Rendering*, second ed. Morgan Kaufmann, San Mateo, CA.

AKIBA, H., MA, K.-L., AND CLYNE, J. 2005. 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)*, 31–39.

BINOTTO, A. P. D., COMBA, J. L. D., AND FREITAS C. M. D. 2003. 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)*, 69–76.

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proc. 4th Symp. Volume Visualization (VVS'94)*, 91–98.

CLYNE, J. 2003. The multiresolution toolkit: Progressive access for regular gridded data. In *Proc. 3rd IASTED Int'l Conf. Visualization, Imaging, and Image Processing (VIIP'03)*, 152–157.

FERNANDO, R., Ed. 2004. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley, Reading, MA.

FOUT, N., MA, K.-L., AND AHRENS, J. 2005. Time-varying, multivariate volume data reduction. In *Proc. 20th ACM Symp. Applied Computing (SAC'05)*, 1224–1230.

GAILLY, J., AND ADLER, M., 2005. zlib. `http://www.zlib.net/`.

GAO, J., WANG, C., LI, L., AND SHEN, H.-W. 2005. A parallel multiresolution volume rendering algorithm for large data visualization. *Parallel Computing 31*, 2 (Feb.), 185–204.

GRAMA, A., GUPTA, A., KARYPIS, G., AND KUMAR, V. 2003. *Introduction to Parallel Computing*, second ed. Addison-Wesley, Reading, MA.

GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. 2002. Interactive rendering of large volume data sets. In *Proc. 13th IEEE Visualization Conf. (VIS'02)*, 53–60.

HADWIGER, M., KNISS, J. M., ENGEL, K., AND REZK-SALAMA, C. 2002. High-quality volume graphics on consumer PC hardware. In *SIGGRAPH 2002, Course Notes 42*.

INO, F., OOYAMA, K., AND HAGIHARA, K. 2005. A data distributed parallel algorithm for nonrigid image registration. *Parallel Computing 31*, 1 (Jan.), 19–43.

IOURCHA, K., NAYAK, K., AND HONG, Z. 1999. System and method for fixed-rate block-based image compression with inferred pixel values, US Patent 5,956,431.

KHAILANY, B., DALLY, W. J., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., TOWLES, B., CHANG, A., AND RIXNER, S. 2001. Imagine: Media processing with streams. *IEEE Micro 21*, 2 (Mar.), 35–46.

LUM, E. B., MA, K.-L., AND CLYNE, J. 2002. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Trans. Visualization and Computer Graphics 8*, 3 (July), 286–301.

MA, K.-L., 2003. Time-Varying Volume Data Repository. `http://www.cs.ucdavis.edu/~ma/ITR/tvdr.html`.

MA, K.-L. 2003. Visualizing time-varying volume data. *Computing in Science and Engineering 5*, 2 (Mar.), 34–42.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graphics 22*, 3 (July), 896–897.

MONTRYM, J., AND MORETON, H. 2005. The GeForce 6800. *IEEE Micro 25*, 2 (Mar.), 41–51.

MOORE, G. E. 1965. Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr.), 114–117.

OBERHUMER, M. F. X. J., 2005. LZO real-time data compression library, Oct. `http://www.oberhumer.com/opensource/lzo/`.

OPENGL EXTENSION REGISTRY, 2004. Gl_ext_texture_compression_dxt1. `http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_dxt1.txt`.

OPENGL EXTENSION REGISTRY, 2004. Gl_nv_texture_compression_vtc. `http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_compression_vtc.txt`.

SCHNEIDER, J., AND WESTERMANN, R. 2003. Compression domain volume rendering. In *Proc. 14th IEEE Visualization Conf. (VIS'03)*, 293–300.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2003. *OpenGL Programming Guide*, fourth ed. Addison-Wesley, Reading, MA.

STRENGERT, M., MAGALLÓN, M., WEISKOPF, D., GUTHE, S., AND ERTL, T. 2005. Large volume visualization of compressed time-dependent datasets on GPU clusters. *Parallel Computing 31*, 2 (Feb.), 205–219.

TAKEUCHI, A., INO, F., AND HAGIHARA, K. 2003. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing 29*, 11/12 (Nov.), 1745–1762.

ZIV, J., AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory 23*, 3 (May), 337–343.