IEEE/EG International Symposium on Volume Graphics (2010)
R. Westermann and G. Kindlmann (Editors)

A GPU-Supported Lossless Compression Scheme for Rendering Time-Varying Volume Data

Jörg Mensmann, Timo Ropinski, and Klaus Hinrichs

Visualization and Computer Graphics Research Group (VisCG),
Department of Computer Science, University of Münster, Germany

Abstract

Since the size of time-varying volumetric data sets typically exceeds the amount of available GPU and main memory, out-of-core streaming techniques are required to support interactive rendering. To deal with the performance bottlenecks of hard-disk transfer rate and graphics bus bandwidth, we present a hybrid CPU/GPU scheme for lossless compression and data streaming that combines a temporal prediction model, which allows to exploit coherence between time steps, and variable-length coding with a fast block compression algorithm. This combination becomes possible by exploiting the CUDA computing architecture for unpacking and assembling data packets on the GPU. The system allows near-interactive performance even for rendering large real-world data sets with a low signal-to-noise-ratio, while not degrading image quality. It uses standard volume raycasting and can be easily combined with existing acceleration methods and advanced visualization techniques.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; E.4 [Coding and Information Theory]: Data Compaction and Compression.

1. Introduction

Improvements in programmable graphics hardware have made interactive volume visualization possible for data from many different domains like medicine or seismology. While the resolution of these data sets is constantly increasing due to advancements in acquisition technology, graphics processors could keep up with the growing amount of data by means of boosting computation performance and graphics memory. GPU-based raycasting [KW03] can easily achieve interactive frame rates for many data sets even without including optimization algorithms. However, this only holds true as long as all data required by the visualization fits completely into GPU memory. When this is not possible, data needs to be streamed from system memory to GPU memory and potentially even from mass storage. The transfer bandwidth has not kept up with the advances in GPU performance, and therefore serious performance degradations must be expected when data sets require out-of-core streaming. Time-varying volume data can easily reach sizes in the range of gigabytes—or even terabytes in the domain of petascale visualization. Time-varying volume data sets can be acquired by medical scanners, although in this domain only a relatively low temporal resolution is used. In contrast, time-varying data sets with high spatial and temporal resolution are routinely created in numerical simulations, especially in the field of fluid dynamics and meteorology.

While the volumetric data resulting from large-scale simulations is often primarily intended for applying statistical analysis models, visualization can be essential for understanding unexpected results or spotting errors. Precomputed animations can solve this problem only to a certain degree, as interactive modification of viewing parameters like camera position and transfer function is generally seen as a necessity for visualization of volumetric data. Additionally, the occlusion problem is more of an issue for volume series than for a static volume, as the camera position may need continuous updates to keep the region of interest in sight.

To allow interactive rendering of large time-varying volumetric data sets, techniques to accelerate data streaming to the GPU must be applied. A common approach is to use data compression techniques in order to reduce the amount of data that needs to be transferred through bandwidth bottlenecks. Many compression techniques have been proposed for static as well as time-varying data sets. However, lossless compression techniques for rendering time-varying data sets are rare, although a domain expert has to be able to access all

© The Eurographics Association 2010.

DOI: 10.2312/VG/VG10/109-116

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

details of the data at a high accuracy. Especially when considering the amount of time spent on the simulation, it is of great interest to be able to inspect the data without any reduction. Therefore we present a lossless compression scheme, which meets these requirements and allows near-interactive frame rates even for large data sets. When visualizing data with our approach, the domain expert can rely on the fact that all visible features are actually present in the data and do not occur due to compression artifacts.

To achieve this goal, our technique utilizes direct programming of the graphics processor through the CUDA programming interface. With this programming functionality, simple compression techniques can be directly brought to the GPU. Since typically the graphics processor is not used to capacity when data needs to be streamed, it has free resources to support this compression, which previously would have been handled exclusively by the CPU. Our hybrid approach allows the combination of several different compression methods, which are optimized for different parts of the data transfer pipeline. Thus, we are able to achieve an efficient lossless compression, which is essential to allow fast streaming. As most components of our approach work independently from each other, the lossless compression scheme can be modified by exchanging individual parts or adding further ones. It can therefore be viewed as a generic framework for combining CPU and GPU techniques to improve data throughput and rendering performance. Since the decompression is performed on-the-fly, the presented approach does not limit the visualization techniques that can be applied to the data.

2. Related Work

There exists a multitude of approaches for compressed volume rendering and for visualization of time-varying volume data sets [Ma03]. They use lossless or lossy compression, or a combination of both. A second distinction can be made between CPU-based, GPU-based, and hybrid CPU/GPU techniques.

Guthe et al. [GWGS02] presented a lossy CPU-based hierarchical wavelet decompression for rendering of very large volume data sets using hardware-accelerated slice rendering. Vector quantization was used by Kraus and Ertl [KE02] in a GPU-based compression scheme for static and time-varying volumetric data sets. Sohn et al. [SBS04] described a compression scheme for encoding time-varying volumetric features to support isosurface rendering. It is based on a lossy wavelet transform with temporal encoding. A block-based transform coding scheme for compressed volume rendering using vector quantization was introduced by Fout and Ma [FM07], which performs decompression on the GPU by rendering to slices of a 3D texture. Nagayasu et al. [NIH08] presented a pipeline rendering system for time-varying volume data. It uses a two-stage CPU/GPU decompression that combines lossy hardware texture compression on the GPU (DXT/S3TC) with lossless LZO compression on the CPU. Because it is based on the simple hardware compression originally intended for 2D textures, the system is limited to 8-bit scalar data and is prone to visual artifacts. While achieving interactive frame rates through a high compression ratio, the authors report visible artifacts that could mislead the user, but assess the image quality as tolerable for time-series analysis.

Several lossless compression algorithms and prediction schemes for volumetric medical data were compared by Ait-Aoudia et al. [AABY06]. While most of these techniques have been developed for static data, Binotto et al. [BCF03] proposed a lossless compression approach using fragment shaders, based on the concept of adaptive texture maps as introduced by Schneider and Westermann [SW03]. It subdivides the volume into 3D blocks and replaces duplicate and homogeneous blocks by references. As this relies on exact matches between blocks, the approach is most effective for sparse data sets with a low noise level, which are hardly found with complex numerical simulation data.

Smelyanskiy et al. [SHC*09] used a slice-based variable-length coding to compress static volume data on the x86 and Larrabee architectures, which they report to be more effective and faster than ZLIB compression. Fraedrich et al. [FBS07] presented an implementation of lossless Huffman coding as a fragment shader that allows to store up to 3.2 times more volume data without loss of information. However, the decoding throughput of this technique lies in the range of the transmission rate of PCI Express, undoing any savings achieved by the data compression. Hence, this result demonstrates the difficulty of porting compression algorithms to a GPU architecture.

3. Hybrid Compression Scheme

3.1. Data properties and hardware limitations

Volume data acquired from medical scanners is typically stored using 12-bit or 16-bit integer values. Simulations, on the other hand, return floating-point data with a highly varying value range. While modern graphics processors directly support 32-bit float textures, 16-bit integer data is generally seen as sufficient for most volume visualization tasks. Direct compression of float data, as described, for example, by Lindstrom and Isenburg [LI06] for integration into a large scale simulation cluster, is beyond the scope of this paper. Hence, we convert the available simulation test data from float to integer format during preprocessing, spreading the values according to the minimum and maximum values found in the data set to make full use of the 16-bit value range. Depending on the actual application, a more elaborate mapping might be needed.

While there exist data sets with extremely high temporal as well as spatial resolution, for many applications a single time step of a time-varying data set can easily fit into

© The Eurographics Association 2010.

data set	resolution	steps	step size
combustion	$448 \times 704 \times 128$	122	77 MB
convection	512×256^2	401	64 MB
hurricane	$512^2 \times 128$	48	64 MB

Table 1: *Properties of the time-varying data sets tested with our approach (using 16-bit integer scalar values).*

graphics memory. Having the complete time step available to the GPU has several advantages for rendering, in contrast to splitting up data, e. g., into individual volume bricks. First, no overhead is introduced for managing and compositing several parts of the volume or for border handling. Second, availability of the complete volume allows us to use visualization and acceleration techniques that require more information than what is available in a single brick. Finally, implementation is greatly simplified. Therefore it is desirable for the decompression to assemble the data back into its original form as a single 3D texture in GPU memory. Hence, as our approach makes the uncompressed data accessible as a standard volume texture during rendering, no multi-pass bricking techniques have to be exploited but standard rendering can be used. In the optimal case a volume rendering system can be extended to support such an out-of-core rendering of time-varying data by just replacing the modules for loading from disk and uploading into a 3D texture, while the actual rendering may stay untouched. This is an important aspect especially in the context of existing large-scale visualization systems. Furthermore our technique can be combined with multi-resolution approaches [LLY06]. Since we employ a lossless compression, a multi-resolution data set can be compressed and streamed by using our approach without affecting its content.

A major reason for the high volume rendering performance achieved by current graphics processors is the graphics memory bandwidth. For example, an NVIDIA GeForce GTX 280 achieves 110 GB/s for an on-device copy. When data needs to be streamed from the CPU to the GPU via the PCI Express bus, the achievable throughput is more than an order of magnitude lower at 2.5 GB/s. Finally, when the data must be read from mass storage, current desktop hard drives achieve around 110 MB/s and server hard drives up to 170 MB/s. Combining several drives can improve throughput, but it is obvious that mass storage is the major bottleneck for streaming data to the GPU.

3.2. Two-stage compression approach

As the size of a single time step for typical time-varying data sets is already in the range of what a hard drive can transfer per second (compare Table 1), it becomes clear that any technique that aims at interactive rendering must minimize the amount of data that needs to be loaded from disk, i. e., it must maximize the compression ratio of the on-disk storage format.

It would be optimal to run a decompression algorithm on the GPU, as the data would then travel through both described bottlenecks in compressed form. Unfortunately, the highly parallel architecture of current GPUs is not well suited for compression tasks. Most algorithms for data compression work in a serial fashion and show no coherent branching behavior, which does not map well to GPUs. Hence, the main decompression must be performed by the CPU. As the bandwidth between CPU and GPU is an order of magnitude greater than that of mass storage, getting maximum compression with this transfer is not as important as when loading from disk. But as decompression can use the CPU to full capacity, moving calculations to the GPU can be beneficial. This is possible only for simple computations that fit into the highly parallel architecture, but even simple memory copy operations can benefit from the higher memory bandwidth available compared to the CPU. Therefore we propose a two-stage or hybrid compression scheme. Data is compressed twice, first with a simple algorithm whose decompression component runs efficiently on the GPU, then with a CPU-based compression technique. Care must be taken that the output of the first compression is still suitable for the second compression step to be effective. On the other hand, an initial compression step that preprocesses data so that they can be compressed more efficiently by the second technique would be useful.

3.3. Subdividing the volume into bricks for compression

Volume data is usually stored with a simple memory layout where the two-dimensional slices that form the volume are saved one after another. Previous work often used 2D compression techniques on these individual slices. Working with slices has the advantage that the memory format is identical to that of the final 3D texture used for rendering, but this comes at the cost of losing spatial coherence. Two voxels which are close together in volume space can actually lie far away in memory space and vice versa. This effect can be evaded by subdividing the volume into three-dimensional bricks and storing the contents of each brick as a continuous block in memory, hence reducing the memory range used per brick. This bricking scheme is only used for compression and data transfer, but not for rendering. Therefore, it does not introduce an overhead to the rendering, but requires the bricks to be assembled back to the original form of a single 3D texture. This is a simple operation that can be run very efficiently on the GPU. While OpenGL supports brickwise updates of 3D textures, it only allows a direct copy with no possibility of data reduction. So even a brick with all zeros would need to be transferred completely. OpenGL also supports slice-based writing to 3D textures from a fragment shader, but this requires considerable overhead and is inflexible. Using CUDA for assembling the bricks back into a complete volume on the GPU is more flexible and allows for better performance.

© The Eurographics Association 2010.

A good opportunity for optimization after the volume has been decomposed into bricks is removing duplicate bricks and replacing them by references [BCF03]. This, however, requires an exact match and is only applicable for data without noise, i. e., mostly synthetic data. Real-world data sets we tested did hardly contain any non-uniform duplicate bricks when choosing a feasible brick size, hence we do not see this method as beneficial for our use case. The only type of duplicate bricks that appears regularly is a uniform brick where all voxels are set to zero. This case is efficiently handled by the variable-length coding described in Section 3.6.

3.4. Main compression algorithm

To choose a compression algorithm for our use case we must take both compression ratio and decompression speed into account. As a requirement, the time for reading and uncompressing a data block must be less than the time that would be needed for reading the uncompressed block. However, most lossless data compression tools and libraries such as gzip or bzip2 are optimized for maximum compression ratio, but not speed. We have evaluated these with parts of the *convection* data set, but none of them was able to decompress faster than it would take to read the uncompressed file. As an alternative suggested by Nagayasu et al. [NIH08], we chose the Lempel-Ziv-Oberhumer (LZO) real-time compression library [Obe08]. It uses the same Lempel-Ziv dictionary coder as gzip, but was built with the main goal of providing fast decompression. The library supports multiple algorithms, from which we selected the LZO1X-999 variant, which yields the best compression ratio. It is the slowest of the available LZO compressors (up to 8 times slower than the default in our tests), but this does not influence the decompression speed.

3.5. Prediction schemes

The block compression algorithms described in the previous section can reduce the size of volume data by utilizing spatial coherence. But they cannot directly take advantage of temporal coherence between different time steps, because their sliding window is not large enough to cover several time steps. To utilize temporal coherence in time-varying data, a prediction model needs to be applied. Such a model tries to predict voxel values and replaces them by the error in the prediction [AABY06, FBS07]. For time-varying data it is promising to predict that the current voxel value will not change in the next time step and to store the difference to the actual value, i. e., the error in the prediction. This differential pulse-code modulation (DPCM) or delta encoding initially does not reduce the storage requirements. However, when the changes between time steps are not random, the error data will exhibit uniform structures. For example, all voxels in regions that do not change between time steps will get a delta value of zero, resulting in uniform bricks that can be compressed efficiently. Also the resulting delta values will typically not use the full data range that is taken up by the original values. Therefore the distribution of delta values will be non-uniform, which allows further compression. A disadvantage of a delta encoding is that it prevents jumping directly to a certain time step, as all previous time steps first have to be read to reconstruct the data. This can be resolved by saving the absolute values in addition to the delta values and loading them on demand, at the cost of increased storage requirements.

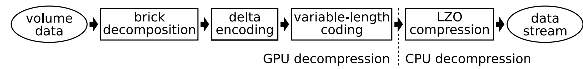


Figure 1: *Block diagram of the complete volume compression scheme, also showing which parts are optimized for decompression on the GPU and on the CPU.*

3.6. Variable-length coding

As values inside a data set are usually not uniformly distributed over the volume, the value range of some bricks will be smaller than the value range of the entire volume. When the difference between the maximum and the minimum value in one brick is less than 2^n with $n < 16$, the brick size can be reduced by storing the minimum as the brick's base value, and for each voxel the difference from the base. Each of the difference values now only takes n bits to store, so this *variable length-coding* reduces the size needed for storing the brick.

This approach can be directly applied in combination with the previously described temporal prediction scheme. As the resulting delta values are typically smaller than the absolute voxel values they encode, the variable-length coding can achieve much higher compression ratios when applied to the delta values compared to when the time steps are compressed using absolute values. Uniform bricks are handled by the variable-length coding directly: All voxels in such a brick have the same value, so just the base value needs to be stored, while the delta values are reduced to “zero bits”, i. e., are omitted.

3.7. Preprocessing and on-disk storage format

The entire data set processing as shown in Figure 1 can run as an offline preprocessing step, with the aim of minimizing the overall data size. The runtime of this preprocessing is usually not an issue, as it is much less computationally intensive than the simulations used for creating the data in the first place. Our preprocessing creates a stream file that contains a sequence of compressed bricks with additional per-brick information, such as the number of bits used for storage. Preprocessing of the test data took 3 minutes for *hurricane*, 23 minutes for *combustion*, and up to 167 minutes for *convection*, with most of the time spent on the LZO compression algorithm. It should be noted, however, that the preprocessing was not optimized for speed, e. g., by running multiple compression threads in parallel.

© The Eurographics Association 2010.

4. GPU-supported Decompression Pipeline

4.1. Multi-threaded loading and LZO decompression

We use a pipeline approach to overlap loading from disk, LZO decompression, and data upload to the GPU (see Figure 2). Each of these tasks runs as one or more independent threads. The loader thread will load up to five time steps in advance and feed them to the decompression thread when it becomes idle. The main bottleneck is typically data loading, so building up of a long queue is only expected for bricks with a high compression ratio, for which loading from disk is faster than decompression.

4.2. Asynchronous data transfer

The data transfer of the uncompressed bricks to the GPU runs through a transfer buffer in main memory that is marked as *page-locked*. Using this "pinned" memory allows us to start an asynchronous memory copy that can run in parallel to CPU operations and GPU kernel executions. It reaches the maximum transfer bandwidth available by copying one large memory block that contains all bricks. While the variable-length coding already handles "empty" (i. e., all-zero) bricks and those bricks that do not change between time steps, the data transfer could be further reduced by ignoring bricks that are completely transparent because of the transfer function. A simple approximation for determining a brick's visibility is comparing the minimum and maximum intensity values inside the brick with the minimum and maximum intensity that is assigned non-zero opacity through the transfer function. This can be implemented efficiently but introduces two issues: First, not loading a brick because it is currently invisible breaks the delta encoding of upcoming time steps, as it requires data from all previous time steps to calculate the current value. Hence, the absolute value would need to be accessible as well, increasing disk usage. The second issue is that when the user modifies the transfer function, bricks that were previously hidden may get visible, therefore requiring a load operation, which might hamper the user experience. In addition, this optimization is not specific to our hybrid compression scheme, so we have not yet implemented it for the current system.

4.3. Brick assembly and resolving prediction

The data packets as uploaded to the GPU require three processing steps before they can be copied to the final 3D texture to be used for rendering: Resolving variable-length coding, resolving delta encoding, and brick assembly.

As the variable-length coding requires different addressing modes based on with how many bits a brick is stored, we have implemented individual kernels for handling each of the supported bit lengths. Based on analysis of our test data, we concluded that the compression ratio for just supporting 16, 8, 4, and 0 bits comes close enough to the optimal result so that the additional costs of supporting all possible numbers of bits are not justified.

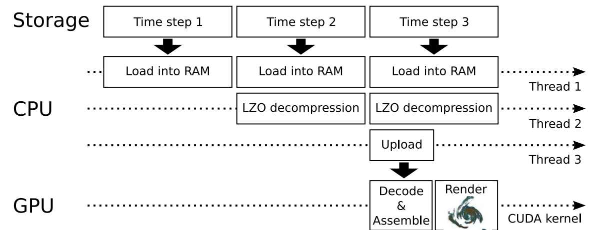


Figure 2: *Our hybrid CPU/GPU decompression pipeline.*

To be able to benefit from the texturing hardware for linear filtering and border handling during rendering, the volume needs to be available to the raycasting kernel as a CUDA array. In contrast to data in global memory a CUDA kernel cannot directly write to such an array. Therefore the decompression kernel uses a shadow copy of the volume texture located in global memory to write its results. The volume is later copied to the CUDA array by calling `cudaMemcpy3D()` from host code. As this is an on-device copy, it can theoretically make use of the full GPU memory bandwidth. This intermediate step is anticipated to become unnecessary with the next generation of graphics processors, which are expected to allow writing to 3D textures from kernel code. The feature is already included in the OpenCL specification through the extension `cl_khr_3d_image_writes`, but this extension is not yet supported by current GPUs and drivers.

Implementing delta encoding is trivial, as the kernel just needs to add the calculated value to the existing value in the volume instead of overwriting it. To obtain optimal performance with CUDA kernels it is most important to satisfy the coalescing rules, i. e., to organize memory accesses so that they require the minimum number of memory transactions. We distribute the assembly of a brick onto blocks of 64 CUDA threads, where each thread is assigned an x-coordinate and processes all voxels in the brick belonging to this x-coordinate. Due to the memory layout, this results in adjacent threads accessing adjacent memory cells and therefore achieving full coalescing. By constructing a suitable two-dimensional CUDA grid of thread blocks, a single kernel launch is enough to start processing of all bricks.

4.4. Rendering

The implementation of GPU-based rendering with CUDA is similar to a fragment shader implementation, with some additional possibilities such as controlling the distribution of rays to threads through the CUDA block size. We have previously investigated raycasting with CUDA [MRH10] and accordingly selected a block size of 8×8 to get optimal results. The raycaster uses direct volume rendering with Phong lighting, on-the-fly gradient calculation, and early ray termination.

© The Eurographics Association 2010.

5. Results

Tests were conducted on a workstation equipped with an Intel Core 2 Quad Q9550 CPU (2.83 GHz), an NVIDIA GeForce GTX 280 GPU, 4 GB RAM, and a 1.5 TB eSATA hard disk with a specified burst transfer rate of 105–115 MB/s. The system was running Linux and used version 3.0 beta of the CUDA Toolkit. We integrated loading and streaming of time-varying data into the *Voreen* volume rendering engine by implementing a single *volume source processor*, making use of the data flow architecture in Voreen.

5.1. Test data sets

Renderings of the test data sets are shown in Figure 3. The *convection* data set is the result of a hydrodynamical simulation of a thermal plume. It contains two modalities, temperature (*T*) and enstrophy (*ens*), where the latter highlights swirling regions of the flow. As can be seen in the first time step of this data set in Figure 3, the *T* modality contains a high level of noise, which was intentionally introduced into the simulation, and ends in a fully turbulent scenario. The *ens* modality is more uniform in the initial part, but also becomes fully turbulent towards the end. Three modalities *chi*, *vort*, and *y_oh* are available from a turbulent combustion simulation. The structure of this data set is turbulent as well, but the amount of empty space varies between the modalities. Finally, there is data from a simulation of the amount of rain in different levels of the atmosphere for Hurricane Isabel. This smaller data set contains many empty regions and is expected to achieve a high compression ratio.

5.2. Compression ratios

To evaluate the effect of the compression parameters, we have compressed *convection/T* with several different options, results are listed in Table 2. Note that the given raw sizes correspond to the data converted to 16-bit, the original float data would take up twice the amount of memory. First, we examined the effect of delta encoding without using variable-length coding. Delta encoding increased the compression ratio of the LZO algorithm from 1.50 to 2.17, which is very significant, considering the low cost of calculating the delta values. The efficiency of variable-length coding depends on the brick size, as smaller bricks are more likely to contain data that fits into a smaller value range. As can be seen from the bit usage, only 4% of the bricks can be encoded with less than the full 16 bits when a brick side length of 256 voxels is used. This percentage increases with smaller brick size, increasing the compression ratio σ_{vlc} achieved by the variable-length coding alone. The best compression ratio is achieved for brick side length 32, but also the number of bricks increases to 1024 for this configuration. To keep the overhead for managing bricks reasonable, we chose a brick side length of 64 for all following tests.

As expected, the compression gave quite different results for the different data sets (Table 3). The modality *T* of the *convection* data set has the lowest compression ratio both for variable-length coding as well as for total compression. This is the result of the high level of noise and low amount of empty space in the data set. The *ens* modality is more sparse and therefore has a much higher compression factor of 5.2, with many more bricks encoded with less than 16 bits. The *combustion* data set contains a lot of empty space, so 41 to 50% of its bricks are empty and can be encoded with zero bits. It is notable that while σ_{vlc} only varies slightly between the modalities, the overall compression factor σ varies between 3.0 and 4.3. Hence, the differences are a result of only the LZO compression. Finally, *hurricane* is a rather small and sparse data set that gets a high compression factor of 25.1 and therefore shifts the system bottleneck from disk throughput to decompression speed.

5.3. Rendering speed

To determine the increase of overall rendering performance achieved by our method, we measured the time taken for rendering all time steps of the compressed data sets and compared this to the results of the uncompressed version (Table 4). The loader for the uncompressed files reads the 16-bit integer data of a time step into memory and immediately uploads it to the GPU. Disk caches were flushed between test runs. The raycasting sampling rate was set to 2 samples per voxel and a viewport size of 512×512 pixels was chosen. As expected, the rendering speedups for the different data sets resemble the compression ratios, for some even slightly exceeding this value. The small *hurricane* data set is not limited by disk throughput and therefore the rendering speedup attained by the compression technique is significantly smaller than the compression ratio. It is rendered with 10 fps, more than 7 times faster than without compression. The larger data sets are also rendered with a near-interactive performance of up to 6 fps.

Measuring the time needed for the on-device copy of the volume from global memory into the final 3D texture stored as a CUDA array (compare Section 4.3) gave results of about 24 ms for the *convection* data set, twice the time needed for brick assembly and even more than the time taken for rendering. This corresponds to a throughput of about 5 GB/s, much less than the maximum of 110 GB/s. We presume that the low throughput for copying to a 3D CUDA array is related to the internal data format, which is not documented by the vendor. Future graphics processors that are expected to allow direct writing to 3D textures from kernels will make this intermediate copy unnecessary.

To examine the efficiency of the GPU-based brick assembly implemented as a CUDA kernel, we compared it to a CPU implementation that assembles the bricks into a memory buffer, which is then uploaded to the GPU. The results in Table 5 show that the CUDA implementation is never slower than the CPU and can achieve a significant rendering

© The Eurographics Association 2010.

								brick bit usage			
pred.	vlc	bs	memory	#bricks	compr. size	σ	σ_{vlc}	16	8	4	0
none	no	64	512 kB	128	16.76 GB	1.50	—	—	—	—	—
delta	no	64	512 kB	128	11.53 GB	2.17	—	—	—	—	—
delta	yes	32	64 kB	1024	11.16 GB	2.24	1.33	61%	18%	20%	1%
delta	yes	64	512 kB	128	11.31 GB	2.22	1.14	78%	15%	6%	0%
delta	yes	128	4 MB	16	11.68 GB	2.15	1.04	93%	7%	0%	0%
delta	yes	256	32 MB	2	11.92 GB	2.10	1.02	96%	4%	0%	0%

Table 2: *Effect of prediction scheme, variable-length coding (vlc), and brick size (bs) on compression, tested with the convection/T data set. Also listed are the total compression ratio σ, compression ratio obtained by variable-length coding alone σ_{vlc}, and the percentages of bricks that are encoded with a certain number of bits by the variable-length coding.*

						brick bit usage			
data set	modality	raw size	compr. size	σ	σ_{vlc}	16	8	4	0
convection	T	25.1 GB	11.3 GB	2.2	1.1	78%	15%	6%	0%
	ens	25.1 GB	4.8 GB	5.2	1.5	52%	19%	12%	17%
combustion	chi	11.4 GB	2.7 GB	4.3	2.0	48%	2%	1%	50%
	vort	11.4 GB	3.8 GB	3.0	1.9	50%	3%	6%	41%
	y_oh	11.4 GB	3.5 GB	3.3	2.0	49%	2%	2%	47%
hurricane	rain	3.0 GB	0.1 GB	25.1	2.6	36%	3%	1%	60%

Table 3: *Results of our hybrid compression scheme. The compression uses 64^3 bricks, delta encoding, variable-length coding, and LZO1X-999 compression.*

		raw		compressed		
data set	mod.	time	fps	time	fps	*s*
convection	T	281	1.4	108	3.7	2.60
	ens	281	1.4	66	6.0	4.23
combustion	chi	126	1.0	28	4.4	4.55
	vort	117	1.0	39	3.1	2.98
	y_oh	125	1.0	36	3.4	3.47
hurricane	rain	35	1.4	5	10.0	7.27

Table 4: *Frame rates for rendering, with time in seconds, frames per second, and speedup factor s.*

		CPU		GPU		
data set	mod.	time	fps	time	fps	*s*
convection	T	117	3.4	108	3.7	1.08
	ens	92	4.4	66	6.0	1.38
combustion	chi	28	4.4	28	4.4	1.01
	vort	39	3.1	39	3.1	1.01
	y_oh	36	3.4	36	3.4	1.00
hurricane	rain	48	6.2	5	10.0	1.29

Table 5: *Efficiency of running brick assembly on the GPU compared to the CPU, s specifies the rendering speedup.*

speedup of up to 1.38 for data sets that are not disk-limited. The CPU implementation writes directly into the 3D texture without an intermediate on-device copy, so the speedup would rise further when the CUDA kernel would also be able to write directly to a 3D texture.

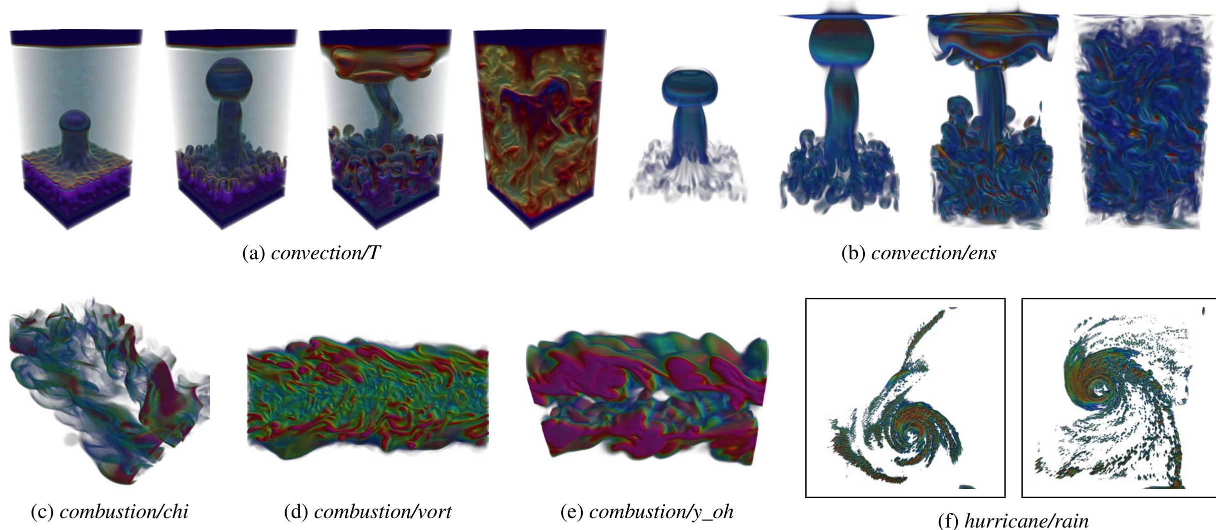
6. Conclusions

In this paper we have presented a framework that allows to increase rendering speed of time-varying data sets based on a lossless compression scheme. By utilizing both CPU and GPU, we could minimize the amount of data that needs to be transferred. Relocating work to the GPU allows us to use more complex prediction models and use brick-based instead of slice-based addressing to better maintain spatial coherence and increase the efficiency of variable-length coding without increasing load on the CPU. While the image quality is not affected by our approach, the compression ratio that can be achieved is highly dependent on the data set. We have demonstrated our technique with real-world data sets that contain a considerable level of noise. In all cases a near-interactive performance was obtained at full image quality, and the compression increases performance so far that fully interactive performance is expected to be achieved when replacing the single hard disk by a faster storage device, e. g., a RAID system.

Since we have exploited recent stream programming techniques, the compression scheme is flexible and can be extended or modified easily. Thus, it would also be possible to integrate lossy compression schemes for application cases where accuracy is not the highest demand. When using the proposed technique, the actual GPU-based volume rendering needs no adaptation and can remain completely unchanged. Therefore, combination with other conventional acceleration techniques, for example, empty-space skipping, is possible. However, for the types of data we tested, the pure raycasting performance on the GPU was not the bottleneck.

Future work includes direct support for floating-point data, evaluating further prediction schemes, and combination with multi-resolution techniques. With new graphics processors it should also be investigated whether they better support the implementation of more complex compression algorithms.

© The Eurographics Association 2010.



(a) *convection/T* (b) *convection/ens*

(c) *combustion/chi* (d) *combustion/vort* (e) *combustion/y_oh* (f) *hurricane/rain*

Figure 3: *Visualization of time steps from the test data sets using direct volume rendering.*

Acknowledgments

This work was partly supported by grants from Deutsche Forschungsgemeinschaft, SFB 656 MoBil (project Z1). The presented concepts were implemented using the Voreen volume rendering engine (http://www.voreen.org). Johannes Lülff and Michael Wilczek from the Institute for Theoretical Physics at the University of Münster provided the convection data set. The turbulent combustion simulation data was made available by Dr. Jacqueline Chen at the Sandia National Laboratory through the SciDAC Institute for Ultra-Scale Visualization. The Hurricane Isabel data was produced by the Weather Research and Forecast Model, courtesy of NCAR, and the U.S. National Science Foundation.

References

[AABY06] AIT-AOUDIA S., BENHAMIDA F.-Z., YOUSFI M.-A.: Lossless compression of volumetric medical data. In *ISCIS* (2006), vol. 4263 of *Lecture Notes in Computer Science*, Springer, pp. 563–571.

[BCF03] BINOTTO B., COMBA J. L. D., FREITAS C. M. D.: Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *PVG '03: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 69–76.

[FBS07] FRAEDRICH R., BAUER M., STAMMINGER M.: Sequential data compression of very large data in volume rendering. In *VMV: Proceedings of the Vision, Modeling, and Visualization Conference* (2007), pp. 41–50.

[FM07] FOUT N., MA K.-L.: Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1600–1607.

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization* (2002), pp. 53–60.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *HWWS: Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2002), pp. 7–15.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization* (2003), pp. 287–292.

[LI06] LINDSTROM P., ISENBURG M.: Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1245–1250.

[LLY06] LJUNG P., LUNDSTRÖM C., YNNERMAN A.: Multiresolution interblock interpolation in direct volume rendering. In *EuroVis: Proceedings of the Eurographics/IEEE Symposium on Visualization* (2006), pp. 259–266.

[Ma03] MA K.-L.: Visualizing time-varying volume data. *Computing in Science and Engineering 5*, 2 (2003), 34–42.

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K.: An advanced volume raycasting technique using GPU stream processing. In *GRAPP: International Conference on Computer Graphics Theory and Applications* (2010). To appear.

[NIH08] NAGAYASU D., INO F., HAGIHARA K.: A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics 32*, 3 (2008), 350–362.

[Obe08] OBERHUMER M. F. X. J.: LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/, 2008.

[SBS04] SOHN B.-S., BAJAJ C., SIDDAVANAHALLI V.: Volumetric video compression for interactive playback. *Computer Vision and Image Understanding 96*, 3 (2004), 435–452.

[SHC*09] SMELYANSKIY M., HOLMES D., CHHUGANI J., LARSON A., ET AL.: Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1563–1570.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proceedings of IEEE Visualization* (2003), pp. 293–300.

© The Eurographics Association 2010.