computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

# MULTIRESOLUTION RENDERING BASED ON GPGPU COMPUTING

**Julián Lamas-Rodríguez, Francisco Argüello, and Dora B. Heras**

CITIUS, University of Santiago de Compostela, Spain
{julian.lamas, francisco.arguello, dora.blanco}@usc.es

**Abstract:** The problem of visualizing large volumetric datasets is appealing for computation on the GPU. Nevertheless, the design of GPU volume rendering solutions must deal with the limited available memory in a graphics card. In this work, we present a system for multiresolution volume rendering which preprocesses the dataset dividing it into bricks and generating a compressed version by applying different levels of compression based on wavelets. The compressed volume is then stored in the GPU memory. For the later visualization process by texture mapping each brick of the volume is decompressed and rendered with a different resolution level depending on its distance to the camera. This approach computes most of the tasks in the GPU, thus minimizing the data transfers among CPU and GPU. We obtain competitive results for volumes of size in the range between $64^3$ and $256^3$. 

**Keywords:** compressed volume rendering, texture mapping, multiresolution rendering, wavelet transform, quantization, CUDA, OpenGL.

## 1. INTRODUCTION

The evolution from graphics-specific accelerators to programmable vector processors has made of GPUs a standard platform for rendering volumetric datasets. However, recent years have witnessed significant improvements in the data acquisition methods, and, as a result, the size of datasets has increased. This poses a challenge given the limited memory resources available in current graphics hardware, and although each new GPU generation expands its memory capacity, the current trend shows that this problem will continue to exist in the future [1]. In this context, compression stands as an effective solution for processing increasingly larger datasets in the GPU. The compression is usually computed on a previously decomposed version of the volume.

As it is usual in the context of volume rendering, the volume is initially decomposed into a set of non-overlapping blocks, usually called bricks, so a single brick fits into the memory of the GPU. Bricks are compressed with multiple levels of compression. When the visualization process begins the bricks are loaded and rendered one at a time. In our implementation we use a single OpenGL 3D texture buffer to store the contents of a brick of data, and this buffer is reused every time a new brick is processed. The final visualization is achieved through texture mapping (also known as texture slicing) [9], which, along with ray casting, is one of the most popular methods to render volume data.

In this rendering technique, the 3D texture is mapped onto a proxy geometry composed of planar polygons that constitute camera-oriented translucent slices, i.e., the volumetric object is cut into slices that are rendered always parallel to the image plane [10, 11].

Several techniques of compressed volume rendering that can be found in the literature rely on storing the compressed volume data in a memory space different from the GPU (as the CPU main memory or a hard disk) [12, 13]. These out of-core techniques require transfer decompressed portions of the volume to the GPU memory before they can be rendered. Their performance is limited, at least, by the transfer rate of the PCI bus (e.g., 8 GB/s for PCIe 2.0).

A wide variety of approaches have been developed to build a compact representation of the data. In volume rendering, these solutions are usually asymmetric, i.e., the original dataset is decomposed and compressed in an off-line process which is executed only once without execution time constraints, while the decompression and visualization processes are executed in real time. Common methods for data compression may involve applying wavelet transforms [2, 3], vector quantization [4-6], or a multiscale tensor approximation [7]. For a survey on compressed

GPU-based volume rendering, we refer the reader to [8].

In this work, we have used a wavelet transform to compress the volume into a compact hierarchical form. We have selected the *Haar* wavelet, as it is computationally simple and very effective for fast reconstruction. Most of the coefficients of this transform are computed as sample-to-sample differences of the original volume data. This means that these coefficients will be of a small magnitude, or even zero, and therefore can be neglected without any significant loss of information. Our encoding scheme, a generalization of [2], benefits from this characteristic to obtain a more compact format of the compressed volume.

In this paper, we present a solution that stores the volume in the GPU memory in its compressed form. We couple decompression and rendering by dividing the volume in bricks which are processed one at a time, benefiting from the higher transfer rate of the GPU memory bus (192.4 GB/s in an NVIDIA GTX 580). Additionally, as our encoding scheme uses a wavelet transform, it supports decompressing bricks at different levels of resolution. The final rendering is executed using the texture mapping technique. We have obtained high speedups for the CUDA implementation of the steps of the algorithm. The complete system performs at an interactive and stable frame rate independent of the viewport size, while keeping a good compression ratio with a high visualization quality. This work is an extension of [9] where the rendering system was presented.

The rest of this paper is organized as follows. Section 2 describes the GPU architecture. Section 3 examines the design of our GPU-based system for compressed volume rendering. Section 4 analyzes the experimental results and compares our implementation to other similar works. Finally, Section 5 concludes discussing the main contributions and future work.

## 2. GPU ARCHITECTURE

GPUs are programmable architectures consisting of several many-core processors capable of running hundreds of thousands of threads concurrently. In this section we present a brief overview on the Fermi GPU architecture [15], which we have used to test our implementation of a GPU volume-rendering system.

NVIDIA's CUDA architecture [16] consists of a huge number of cores (or streaming processors, SPs), grouped into a set of streaming multiprocessors (SMs), with a very high memory bandwidth. As an example of this architecture, the GeForce GTX 580 has 16 SMs with 32 SPs each, resulting in 512 cores.

The programming model encourages a fine-grained level of parallelism within the single program multiple data (SPMD) paradigm [17]. A CUDA program (called *kernel*) is run by a *grid of threads*, which are grouped in *thread blocks*. Programmers can configure the size and distribution of the grid to their convenience and according to the requirements of the tasks to compute.

The architecture features several memory spaces. The *global memory* and *texture memory* spaces are accessible by the GPU, and also by the CPU through the PCI bus. Other memory spaces are located inside the chip, and provide a much lower latency: a read-only *constant memory*, a *shared memory* (which is private for each SM), a *texture cache* and, finally, a two-level cache that is used to speed up accesses to the global memory.

Coordination between threads within a kernel is achieved through *synchronization barriers*. However, as thread blocks run independently from all others, their scope is limited to the threads within the thread block.

## 3. THE RENDERING SYSTEM

Our solution involves two different stages: compression and visualization. The compression is executed on the CPU to preprocess the data generating the compressed volume from the original dataset.

The visualization stage runs on the GPU, and shows on the screen the reconstructed volume with different resolution levels depending on the distance to the camera. The visualization stage consists of two steps: a reconstruction of the volume followed by the rendering itself. This process is performed brick by brick with the required level of decompression for each brick.

Fig. 1 shows the different data structures used in this implementation. The original volume data is divided into bricks and each brick is divided into blocks of 16 × 16 × 16 elements. In the example shown in the figure a brick contains 2 × 2 × 2 blocks. Each block is divided into cells, each cell containing 4 × 4 × 4 elements. Finally, a chunk contains a group of 2 × 2 × 2 elements.

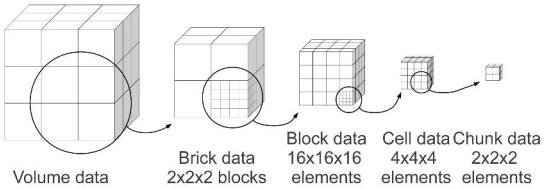


**Fig. 1 – Data structures used in the rendering system.**

Our compression algorithm requires reorganizing the volume data into smaller structures called

blocks, cells and chunks. For example, the wavelet transform that is applied to blocks processes data in a chunk basis. During the visualization stage, the volume is considered to be divided in bricks, which are processed individually.

The different steps of the initial compression and the later visualization stages will be described in the next subsections.

## 3.1. Compression

The compression stage is the preprocessing that takes place before the visualization process. Fig. 2 shows the different steps performed during this stage, and the data generated at each step. First, a wavelet transform is applied to the volumetric data. Afterwards, the wavelet coefficients are quantized to restrict the values to a limited number of possibilities. The quantization step scales down the coefficients obtained by the wavelet transform, nullifying those with a close-to-zero value, so losing information. Finally, the encoding step generates the compressed volume data, which is stored later in the hard disk. All these steps are executed on the CPU.

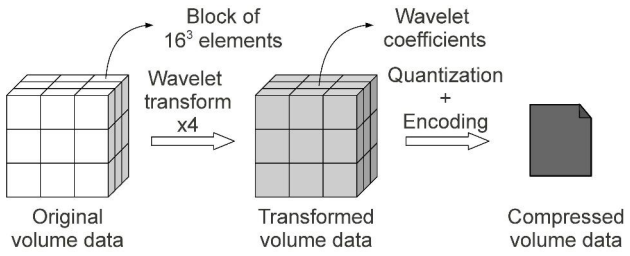


**Fig. 2 – Compression steps over the original volume.**

The wavelet transform step applies a wavelet-transform operator to blocks of 16 × 16 × 16 elements using a Haar filter. Our CPU implementation is similar to other solutions that can be found in the literature [2]. The transform is recursively applied to each block, generating bands of coefficients.

Fig. 3 shows how the wavelet transform generates the coefficients for a 16 × 16 × 16 block, which are then grouped in eight bands. These bands are labeled from LLL to HHH.

The LLL band contains the average coefficients, and the detail coefficients are stored in the remaining bands. The transform is recursively applied to the LLL band, generating new levels of subbands until we get four levels of transform. These levels are the basis of the multiresolution system.

To avoid any data loss during the wavelet application the coefficients are preserved without modifications. This means that the magnitude of the coefficients (specially the low-frequency ones) grows each time the transform is applied. This approach increases the storage requirements but guarantees that the only source of data loss is in the later quantization step.

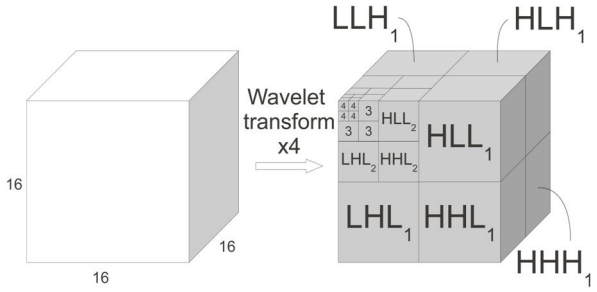


**Fig. 3 – The result of applying a 4-level wavelet transform to a 16 × 16 × 16 block of data.**

Quantization is a lossy compression technique that reduces the range of the values of the compressed dataset [18]. In our implementation we have chosen a scalar quantization solution with fixed-rate coding that removes the least significant bits of the coefficients obtained from the previous wavelet transform. This quantization reduces the magnitude of the coefficients, and nullifies those with a close-to-zero value. The quantization level must be decided according to the compression quality, where not only the compression ratio, but also the signal to noise ratio are considered.

The encoding step converts the resulting volumetric data from the wavelet-transform and quantization steps into its final compressed form following a compromise between good compression ratio and fast random access.

Fig. 4 shows the main data structures used in this step and their meaning. The cell-tag table array stores a cell-tag table for each block in the volume. A cell-tag table contains two-byte tags labeling each cell in a block. The most significant byte stores the width in bytes of the coefficients in the cell (or zero for a null cell), and the less significant byte stores the index of the significance map for the cell. The significance map array contains a bitmap for each non-null cell in the volume. This bitmap is used to flag coefficients in the cells as zero or nonzero. Although it has not been represented in Fig. 4, it is also necessary to store an offset value for each cell. It is stored in 8 bytes (long integer) and contains the corresponding non-null coefficient array.

Finally, four arrays store all the non-null coefficients from the transformed volume data. Each coefficient is stored in an array depending on its width in bytes. This encoding supports coefficients of up to four bytes.

Our encoding solution increases the flexibility of the implementation presented in [2], which was

limited to 4 × 4 × 4-cell blocks. Two-byte cell tags enable using bigger blocks, so more resolution levels could be supported, as the maximum number or recursive wavelet transforms that can be applied is restricted by the size of the block.

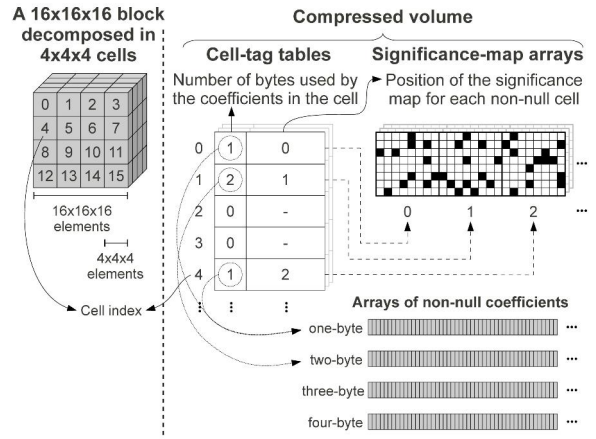


**Fig. 4 – Structure of the encoded data.**

## 3.2. Visualization

The visualization stage is responsible of reconstructing the compressed volume on the GPU (decoding and inverse wavelet transform computation) and rendering it on the screen. Fig. 5 shows the different steps that take place in this stage. The volume is processed brick by brick.

First, a brick is selected from the compressed volume and reconstructed at a specific level of resolution. This reconstruction involves the steps of decoding and inverse transform, which have been implemented in CUDA kernels, and hence, run on the GPU. The restored brick data are stored in an OpenGL Pixel Buffer Object (PBO), and then copied into a texture buffer to be mapped onto a proxy geometry. These operations including the final rasterization are implemented using the OpenGL API. The process continues with another brick until the complete volume has been rendered.

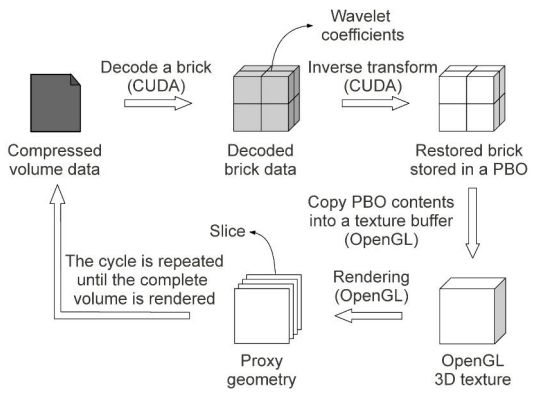


**Fig. 5 – Visualization system for decompressing and rendering the volume data.**

The visualization process is performed brick by brick. In each frame, the CPU decides in which order bricks should be reconstructed according to the position of the camera. A back-to-front order is maintained to guarantee a correct composition of the bricks.

For each brick, depending on its distance to the camera, the CPU chooses a resolution level. Bricks that are close to the camera are rendered at the highest level. Bricks that are far from the camera do not contribute to the final result as much as the closer ones, so in order to speed up the whole process they are rendered at a lower level of resolution.

As stated earlier, two CUDA kernels execute the steps of decoding and inverse transform required to decompress the brick data in order to reconstruct the whole volume. The decompressed data are stored in a PBO, which can be accessed by the CUDA and OpenGL functions.

To complete the visualization, an OpenGL call copies the brick data from the PBO into a texture buffer. Then, the CPU orders the construction of the proxy geometry using several OpenGL calls. This proxy geometry contains the slices where the brick texture is mapped onto.

Depending on the resolution level, the texture might not completely fill the available space in the texture buffer. That is, the highest resolution level uses the complete texture space, but low-resolution textures require only a small portion of that space. This means that the texture coordinates assigned to each vertex of the proxy geometry must be adjusted to the real texture size according to the resolution level chosen for the current brick.

Analyzing each step of the visualization stage more in detail, first we have to pay attention to the decoding step. The process of decoding is performed in a kernel on the GPU. This kernel reads the compressed data of the brick from the compressed volume stored in the GPU global memory and writes the decoded data in a previously allocated buffer (to be later processed by the inverse wavelet transform). Each data block in the brick is assigned to a thread block, where each thread processes a cell (whose size is 4 × 4 × 4 in our implementation).

The decoding process is as follows. Each thread starts by determining if its cell is non-null or not, as indicated by the cell tag associated to the cell. If the cell is non-null, the data reconstruction begins. The thread loops through the elements of the cell, and tries to load them from the arrays of non-null coefficients in the compressed volume (see Fig. 4) accessing the information stored in the significance-map of the cell and considering the offset value for the cell. If the cell's significance map identifies a

coefficient as non-null, its value is stored as is in the buffer in global memory, otherwise a zero is written.

For each brick, data from the different blocks are initially interleaved in global memory attending to their absolute position in the brick. In order to increase the spatial locality of memory accesses, the decoded data are contiguously stored in global memory in a blockwise fashion. Our proposal arranges the data of each block together to reduce the time spent in memory accesses when the inverse wavelet transform is computed.

Once the decoding kernel has finished, another kernel performs an inverse wavelet transform on the GPU to restore the brick contents. Each data block in the brick is assigned to a thread block depending on its identifier, and each thread processes a chunk of 2 × 2 × 2 voxels although this could be modified with minor changes in the implementation.

The inverse transform is a recursive process, and it is applied until the desired level of resolution is achieved. The resulting coefficients of processing a level of resolution are stored in shared memory, where this data will be available to compute the next level of resolution. When the desired level is reached, these coefficients are copied from shared memory into the OpenGL PBO. In the case of processing the highest level of resolution, the coefficients are directly stored in the PBO, bypassing the shared memory and consequently reducing its impact on the memory load.

When storing data in the PBO, the positions where data are placed are determined by the identifiers of the thread block and the current resolution level. For low resolution levels, the data generated by each thread block are grouped in order to avoid chunks of data scattered in the PBO.

Fig. 6 shows how a 32 × 32 × 32 restored brick is stored in the PBO for different levels of resolution.

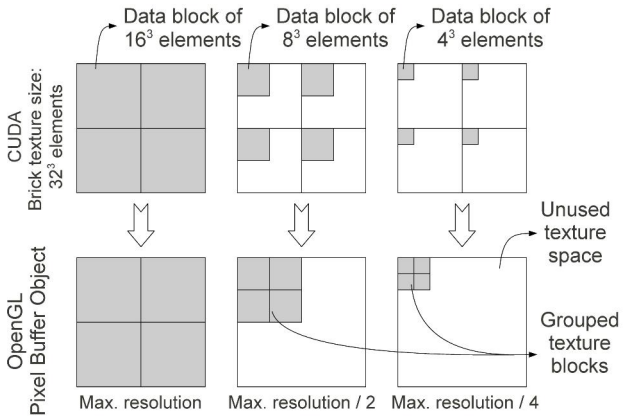


**Fig. 6 – Storing data from shared memory into the PBO for different resolutions.**

# 4. RESULTS

## 4.1. NUMERICAL RESULTS

We performed our tests on a machine consisting on a CPU multicore and a GPU. The CPU is an Intel Core 2 Quad Q9450 with four cores at 2.66 GHz and 6 GB of RAM. The GPU is a NVIDIA GeForce GTX 580 with 16 SMs of 32 SPs each featuring a total of 512 processor cores operating at a clock rate of 1.544 GHz, and with 1.5 GB of global memory. Each SM has 64 kB of RAM with a configurable partitioning of shared memory and L1 cache (16 kB of shared memory and 48 kB of L1 cache, or vice versa). Additionally, a unified L2 cache of 768 kB is available for all SMs [16].

We compiled the code using the NVIDIA nvcc compiler provided within the CUDA 4.0 toolkit and the gcc version 4.4.3 under Linux.

Table 1 details the different datasets used in this work that can be considered as representative instances of volumes obtained from organic tissues and synthetic materials The *BrainWeb* dataset was obtained at the BrainWeb Simulated Brain Database [19]. *ModelHead* corresponds to a volumetric CT of a synthetic model of the human head, whereas *RealHead* is volumetric dataset of a real human head obtained with an MRI technique. The last two volumes, A80 and Knee-001, were obtained through segmentation using a GPU-accelerated level-set segmentation algorithm on two datasets comprising contrast-enhanced CT images. In the case of A80 the dataset corresponds to several brain vessel images that presented some observable cases of aneurysms. Knee-001 corresponds to a knee image. Fig. 7 shows renderings from the datasets using our solution.

**Table 1. Datasets used in this work considering that in all the cases the number of bytes per voxel is 2.**

| Name | Size | File Size |
|---|---|---|
| RealHead | 160 × 512 × 512 | 80 MB |
| Brainweb | 256 × 256 × 181 | 23 MB |
| ModelHead | 512 × 512 × 348 | 174 MB |
| A80 | 512 × 512 × 512 | 256 MB |
| Knee-001 | 256 × 256 × 256 | 32 MB |

## 4.2. QUALITY ANALYSIS AND STORAGE REQUIREMENTS

We have measured the quality of the proposed decoding implementation with the volumes described in Table 1. Tables 2 and 3 show the values obtained for the *BrainWeb* and *ModelHead* datasets for different levels of quantization.

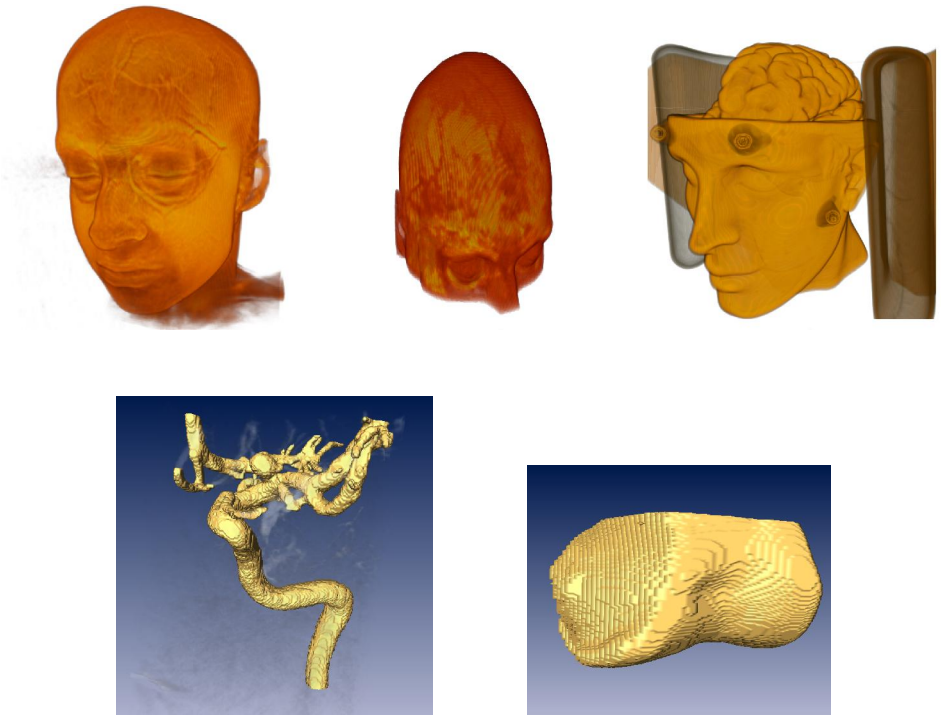

**Fig. 7 – Volume rendering of the test datasets. From left to right and from top to bottom: *RealHead, BrainWeb*, *ModelHead, A80* and *Knee-001.***

Quality is measured here in terms of compression ratio, mean squared error (MSE) and peak signal-to-noise ratio (PSNR). A quantization level corresponds to removing a specific number of least significant bits from the coefficients of the wavelet transform (see Section 3.1.2). Eleven different quantization levels are considered in the experiments, as indicated in the table.

Generally, a value of PSNR above 60 is considered good, so we have chosen a quantization level of 8 bits in our tests to measure performance of the complete GPU volume rendering system (see below). We noticed that changing the number of bits removed during quantization did not significantly affect the performance measured in terms of execution times, however it does severely affect the storage requirements as it was explained in Section 4.1.

The compressed volume is stored in different data structures that were detailed in section 3.2 and shown in Fig. 4. Considering the implementation issues it can be concluded that the nine integer arrays that will be detailed in the next paragraphs are required in order to store the compressed volume.

The cell tags for the cells in the volume are stored in a 2-byte integer array. In addition, for each non-null cell a significance map is generated, so the indices of the maps, the maps themselves and the offsets to access the coefficients are stored in three 8-byte integer arrays.

**Table 2. Compression quality for different levels of quantization for the BrainWeb dataset.**

| # bits | Compr. volume size (MB) | Compress. ratio | MSE | PSNR |
|---|---|---|---|---|
| 0 | 25.52 | 1 : 0.89 | 0.00 | ∞ |
| 1 | 25.10 | 1 : 0.90 | 0.45 | 99.77 |
| 2 | 18.92 | 1 : 1.20 | 0.72 | 97.75 |
| 3 | 16.81 | 1 : 1.35 | 2.28 | 92.76 |
| 4 | 14.85 | 1 : 1.52 | 8.84 | 86.86 |
| 5 | 13.25 | 1 : 1.71 | 35.34 | 80.85 |
| 6 | 11.41 | 1 : 1.98 | 137.92 | 74.93 |
| 7 | 8.63 | 1 : 2.62 | 509.06 | 69.26 |
| 8 | 5.57 | 1 : 4.06 | 1614.11 | 64.25 |
| 9 | 2.89 | 1 : 7.83 | 3488.04 | 60.90 |
| 10 | 1.81 | 1 : 12.50 | 6628.75 | 58.12 |
| 11 | 1.15 | 1 : 19.67 | 12135.79 | 55.49 |

All the non-zero coefficients are stored in four byte arrays. These arrays contain as many bytes as required by the non-null coefficients of different lengths.

Finally one more 8-byte pointer array per brick is required in order to store the indices of the non-null blocks in the brick.

**Table 3. Compression quality for different levels of quantization for the ModelHead dataset.**

| # bits | Compr. volume size (MB) | Compress. ratio | MSE | PSNR |
|---|---|---|---|---|
| 0 | 111.82 | 1 : 1.56 | 0.00 | ∞ |
| 1 | 89.08 | 1 : 1.95 | 0.45 | 99.79 |
| 2 | 64.35 | 1 : 2.70 | 0.63 | 98.31 |
| 3 | 45.87 | 1 : 3.79 | 1.34 | 95.05 |
| 4 | 33.02 | 1 : 5.27 | 3.25 | 91.21 |
| 5 | 23.95 | 1 : 7.27 | 8.58 | 87.00 |
| 6 | 17.74 | 1 : 9.81 | 23.81 | 82.56 |
| 7 | 13.09 | 1 : 13.29 | 66.71 | 78.09 |
| 8 | 9.57 | 1 : 18.18 | 187.64 | 73.60 |
| 9 | 7.08 | 1 : 25.48 | 504.59 | 69.30 |
| 10 | 5.29 | 1 : 32.89 | 1320.25 | 65.12 |
| 11 | 3.82 | 1 : 45.55 | 3117.75 | 61.39 |

For the ModelHead image in Table 1 whose size is 512 × 512 × 348 voxels with 2 bytes/voxel, i.e. a volume of 174 MB, and considering a brick size of 128 × 128 × 128, 16 × 16 × 16 blocks and 4 × 4 × 4 cells, and a quantization level of 8 bits, the size required to store the compressed volume is 9.57 MB. This size could be reduced, as it can be observed in Table 3, if a higher level of quantization is selected, thus loosing quality in the visualized image.

## 4.3. PERFORMANCE ANALYSIS

In order to evaluate the performance we focus on the GPU implementation of the different steps of the rendering system. In particular, we have measured execution times and speedups of the decoding and inverse-transform kernels compared to the CPU implementations. Then, we executed the complete system on the GPU and took measures of execution time for each step and of FPS for the whole system varying the volume size and brick size parameters.

Regarding the speedup measurements, we focus on the implementations of the decoding and the inverse-transform steps implemented in CUDA. Table 4 shows the results we have obtained for different volume sizes that were constructed from the *RealHead* dataset and considering average values for only one brick. High speedups are obtained for both kernels, especially for the inverse transform. For both algorithms, the speedup increases with the volume size, as the computational capabilities of the GPU are better exploited when the number of working threads is larger.

We also evaluate the performance of the whole visualization process on GPU showing the execution times for each step and the frames per second (FPS) obtained. Table 5 shows the performance for two of the datasets whose sizes are described in Table 1 using different brick sizes. As for the other experiments the total time per brick is calculated and multiplied by the number of non-null bricks (the null ones do not require computations) obtaining the total time ("Total" in the Tables). The FPS value is directly calculated from this value. From Table 1, we see that, in general, the larger the brick size, the better the performance obtained. Generally, incrementing the brick size increases the time required to process a brick, but reduces the number of bricks, resulting in a lower time to complete a frame.

Table 6 details results for the *A80* and *Knee-001* datasets. In these cases there are empty bricks in the volumes, i.e. bricks that do not require computation. So, the number of non-null bricks are also specified in the table and considered in the computation of the time per frame ("Total" in the Table). As in Table 5, for the same volume bigger bricks sizes require smaller number of bricks and, therefore, smaller execution times and higher FPS rates.

**Table 4. Execution times in seconds and speedups respect to the CPU implementation of the decoding and inverse-transform kernels operating on a single brick of the *RealHead* image for different brick sizes.**

| Kernel | | 64 × 64 × 64 | 128 × 128 × 128 | 256 × 256 × 256 |
|---|---|---|---|---|
| Decoding | GPU | 0.000077 | 0.000196 | 0.001205 |
| | CPU | 0.003173 | 0.024237 | 0.207878 |
| | Speedup | 41.2x | 123.7x | 172.5x |
| Inverse Transf. | GPU | 0.000027 | 0.000192 | 0.001526 |
| | CPU | 0.009205 | 0.069018 | 0.557935 |
| | Speedup | 340.0x | 352.6x | 365.6x |

**Table 5. Execution times (seconds) and calculated FPS for the steps of the GPU rendering system varying the brick size and considering the *RealHead*, *BrainWeb* and *ModelHead* volumes. All the bricks are non-null.**

| Dataset | Brick Size | # of bricks | Decode per brick (CUDA) | Inv. Transf. per brick (CUDA) | Copy per brick (OpenGL) | Render per brick (OpenGL) | Total per brick | Total | FPS |
|---|---|---|---|---|---|---|---|---|---|
| *RealHead* | $64^3$ | 192 | 0.000077 | 0.000027 | 0.000016 | 0.000382 | 0.000502 | 0.0963 | 10 |
| | $128^3$ | 32 | 0.000196 | 0.000192 | 0.000023 | 0.000699 | 0.001110 | 0.0355 | 28 |
| | $256^3$ | 4 | 0.001205 | 0.001526 | 0.000038 | 0.001546 | 0.004315 | 0.0172 | 57 |
| *BrainWeb* | $64^3$ | 48 | 0.000081 | 0.000027 | 0.000016 | 0.000726 | 0.000850 | 0.0408 | 24 |
| | $128^3$ | 8 | 0.000226 | 0.000191 | 0.000023 | 0.001188 | 0.001628 | 0.0132 | 75 |
| | $256^3$ | 1 | 0.001571 | 0.001534 | 0.000038 | 0.002282 | 0.005425 | 0.0054 | 184 |
| *ModelHead* | $64^3$ | 384 | 0.000075 | 0.000027 | 0.000016 | 0.000529 | 0.000647 | 0.2484 | 4 |
| | $128^3$ | 48 | 0.000215 | 0.000192 | 0.000023 | 0.000940 | 0.001369 | 0.0657 | 15 |
| | $256^3$ | 8 | 0.001206 | 0.001524 | 0.000037 | 0.001559 | 0.004326 | 0.0346 | 29 |

**Table 6. Execution times (second)s and calculated FPS for the steps of the GPU rendering system using *A80* and *Knee-001* datasets and varying the brick size. The number of non-null bricks is also specified.**

| Dataset | Brick Size | # of bricks/ non-null bricks | Decode per brick (CUDA) | Inv. Transf. per brick (CUDA) | Copy per brick (OpenGL) | Render per brick (OpenGL) | Total per brick | Total | FPS |
|---|---|---|---|---|---|---|---|---|---|
| *A80* | $32^3$ | 4096/531 | 0.000062 | 0.000011 | 0.000013 | 0.000246 | 0.000332 | 0.176 | 5 |
| | $64^3$ | 512/129 | 0.000055 | 0.000009 | 0.000016 | 0.000451 | 0.000531 | 0.068 | 14 |
| | $128^3$ | 64/31 | 0.000060 | 0.000023 | 0.000023 | 0.000819 | 0.000925 | 0.029 | 34 |
| | $256^3$ | 8/7 | 0.000139 | 0.000088 | 0.000037 | 0.001565 | 0.001829 | 0.013 | 76 |
| | $512^3$ | 1/1 | 0.000819 | 0.000564 | 0.000061 | 0.002624 | 0.004068 | 0.004 | 250 |
| *Knee-001* | $32^3$ | 512/115 | 0.000060 | 0.000012 | 0.000013 | 0.000431 | 0.000516 | 0.059 | 16 |
| | $64^3$ | 64/29 | 0.000048 | 0.000013 | 0.000017 | 0.000806 | 0.000884 | 0.026 | 38 |
| | $128^3$ | 8/8 | 0.000058 | 0.000030 | 0.000023 | 0.001531 | 0.001642 | 0.013 | 76 |
| | $256^3$ | 1/1 | 0.000240 | 0.000226 | 0.000037 | 0.002981 | 0.003484 | 0.003 | 333 |

## 4.4. COMPARISON TO OTHER WORKS

To the best of our knowledge, this is the first GPU implementation of a decompression scheme based on [2].

The authors reported their solution required, at best, nearly 10 seconds to reconstruct a volume of 512 × 512 × 512 elements on CPU. This includes both the decoding step and the inverse transform step. For a brick of the same size, Table 4 shows a performance between 15 and 20 milliseconds for both steps on the GPU.

Our inverse wavelet transform compares favorably with other GPU implementations in the literature. In a recent work [20], the performance of a 3D fast wavelet transform was measured on a GPU processing 64 frames of a video at different resolutions, requiring 6.8 ms for a 512 × 512 video, and 13.4 ms for a 1024 × 1024 video. This implementation performed a one-level transform using a Daubechies D4 wavelet [21]. To compare these results, we have measured the performance of our inverse-transform kernel for a single level instead of four. Processing a brick of size 256 × 256 × 256, which is exactly the same size as the former video, requires 1 ms in our solution. A 512 × 512 × 512 brick, which is twice the size of the latter video, requires 7 ms.

The performance of the GPU decompression and rendering system is also competitive with similar solutions in the literature. A scheme based on the Karhunen-Loève transform [22] is presented in [1]. Compression is performed on CPU using a vector quantization approach that preserves the coefficients from blocks containing the most relevant edges. Visualization is achieved in a two-pass render, the first one devoted to decompress several slices of data, and the second one to the actual rendering. A 512 × 512 × 512 is rendered at a rate between 6 and 11 FPS, depending on the size of the viewport. For a volume with a similar size (*ModelHead*), our solution achieves 29 FPS without the size of the viewport affecting significantly.

Finally, a solution based on the S3 texture compression algorithm (also known as DXT) [23] was introduced in [24] for time-varying 3D datasets. The reconstruction of the compressed volume data is embedded into a programmable shader, and up to three frames are compressed into the RGB channels of a texture. The authors show results for a volume of size 400 × 600 × 400 visualized at 35 FPS. Although this performance is slightly higher than our solution's, our compression scheme provides better results in terms of quality, with a greater PSNR for a similar compression ratio.

## 5. CONCLUSIONS

In this work we have presented a GPU solution for decompressing and visualizing 3D datasets using a multiresolution rendering scheme. A previous compression stage based on wavelets, and performed in the CPU, is required. The selection of the quantization level applied to the wavelet coefficients is the factor that decides the compression rate and the SPNR value of the compressed volume. A tradeoff value of 8 bits is selected for the quantization level.

The GPU stores the compressed version of the original volume. Our GPU solution processes the compressed volume in 3D data pieces called bricks. For each brick a level of resolution is selected depending on its distance to the camera, and the brick data are decompressed up to that level.

The decompression involves decoding and computing the inverse wavelet transform of the data. Both steps are implemented in CUDA, so they are executed within the GPU. Unlike other out-of-core techniques, communication between CPU and GPU is minimal, avoiding the bottleneck that the PCI bus between both is. As we apply four levels of wavelet, our approach supports up to four different levels of resolution (five including the original one).

The visualization is carried out using the texture mapping technique. The decompressed brick data is copied into an OpenGL texture buffer and mapped onto a proxy geometry composed of several parallel polygonal slices. The GPU rasterizes the geometry by blending the slices to produce the final image.

The solution has been tested with five medical datasets obtaining competitive results compared to other recent GPU implementations of compressed volume rendering. The refresh rates obtained are competitive, the PSNR values are greater than 60, and a compression ratio between 1:4 and 1:18 for volume sizes in the range between $64^3$ and $256^3$ is obtained. A higher quantization level, that could give enough quality for some applications, would increase the compression rate of the solution at the cost of worsening these quality parameters.

As future work, we plan to extend our solution to larger datasets, including datasets that do not fit inside the GPU memory. For these cases, empty-space–skipping techniques are essential to identify bricks in the volume that do not add essential information to the final rendering in order to keep an interactive refresh rate.

## 6. ACKNOWLEDGEMENTS

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, and FEDER funds under contract TIN 2010-17541, and by the Xunta de Galicia under contracts 08TIC001206PR and 2010/28. Julián Lamas-Rodríguez acknowledges financial support from the Ministry of Science and Innovation, Government of Spain, under a MICINN-FPI grant.

## 7. REFERENCES

[1] N. Fout and K.-L. Ma. Transform coding for hardware-acceleratedvolume rendering, *IEEE Transactions on Visualization and Computer Graphics*, (13) 6 (2007), pp. 1600-1607.

[2] I. Ihm and S. Park, Wavelet-based 3D compression scheme for interactive visualization of very large volume data, *Computer Graphics Forum*, Lake Tahoe, CA, (18) 1 (May 2-5, 1999), pp. 3-15.

[3] S. Guthe, M. Wand, J. Gonser, and W. Straßer, Interactive rendering of large volume data sets, in *IEEE Visualization 2002*, Boston, Massachusetts, (Oct. 27-Nov. 1, 2002), pp. 53-60.

[4] J. Schneider and R. Westermann, Compression domain volume rendering, in *IEEE Visualization 2003*, Seattle, Washington, (Oct. 19-24, 2003) pp. 293-300.

[5] R. Parys and G. Knittel, Giga-voxel rendering from compressed data on a display wall, *Journal of WSCG*, (17) 13 (2009), pp. 73-80.

[6] E. Gobbetti, J. A. Iglesias Guitián, and F. Marton, COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks, *Computer Graphics Forum*, (31) 3-4 (2012), pp. 1315-1324.

[7] S. K. Suter, J. A. Iglesias Guitián, F. Marton, M. Agus, A. Elsener, C. P. Zollikofer, M. Gopi, E. Gobbetti, and R. Pajarola. Interactive multiscale tensor reconstruction for multiresolution volume visualization, *IEEE Transactions on Visualization and Computer Graphics*, (17) 12 (2011), pp. 2135-2143.

[8] M. B. Rodríguez, E. Gobbetti, J. I. Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter, A survey of compressed GPU-based direct volume rendering," *Eurographics 2013,* Girona, Spain, *(May 6-10, 2013)*, pp. 117-136.

[9] Julián Lamas-Rodríguez, Francisco Argüello, Dora B. Heras, "A GPU-based Multiresolution Pipeline for Compressed Volume Rendering", *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, EEUU, (July 22-25, 2013), pp. 523-529.

[10] A.V. Gelder and K. Kim, Direct volume rendering with shading via three-dimensional textures, *1996 Symposium on Volume Visualization*, (Oct. 28-29, 1996), pp. 23-30.

[11] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf, *Real-time volume graphics*. A K Peters, Ltd., 2006.

[12] Q. Zhang, R. Eagleson, and T. M. Peters. Volume visualization: a technical overview with a focus on medical applications, *Journal of Digital Imaging*, (24) 4 (2011), pp. 640-664.

[13] E. Gobbetti, F. Marton, and J. A. I. Guitián. A single-pass GPU raycasting framework for interactive out-of-core rendering of massive volumetric datasets, *The Visual Computer,* (24) 7–9, (2008), pp. 797-806.

[14] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash. Octree rasterization: accelerating high-quality out-of-core GPU volume rendering, *IEEE Transactions on Visualization and Computer Graphics*, (99) (2012), pp. 1-14.

[15] J. Nickolls and W. J. Dally. The GPU computing era, *IEEE Micro*, (30) 2 (2010), pp. 56-69.

[16] *CUDA C programming guide (version 4.0), NVIDIA, 2011.*

[17] D. B. Kirk and W.-m. W. Hwu, *Programming massively parallel processors: a hands-on approach.* Burlington, Massachusetts, USA: Elsevier, 2010.

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

[19] C. Cocosco, V. Kollokian, R.-S. Kwan, and A. Evans, BrainWeb: online interface to a 3D MRI simulated brain database, *NeuroImage*, (5) 4 (1997), p. S425.

[20] G. Bernabé, G. D. Guerrero, and J. Fernández, CUDA and OpenCL implementations of 3D fast wavelet transform, *3rd IEEE Latin American Symposium on Circuits and Systems*, Playa del Carmen, Mexico, (Feb. 29- March 2, 2012), pp. 1-4.

[21] I. Daubechies, *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1992.

[22] R. D. Dony, *Karhunen-Loève transform*, in The Transform and Data Compression Handbook. Boca Ratón, Florida, CRC Press, 2004.

[23] K. I. Iourcha, K. S. Nayak, and Z. Hong, *System and method for fixed-rate block-based image compression with inferred pixel values*, US Patent 5 956 431, Sept. 21, 1999.

[24] Y. Cao, L. Xiao, and H. Wang, "Hardware-accelerated volume rendering based on DXT compressed datasets," *International Conference on Audio, Language and Image Processing*, Shangai, China, (Nov. 23-25, 2010), pp. 523-52.

***Julián Lamas Rodríguez,*** *received his B.Sc in Computer Engineering from the University of Coruña, Spain, and his M.Sc in Videogame Creation from the University Pompeu-Fabra in Barcelona, Spain, both in 2006. In 2007 he joined the Systems Laboratory Group in the Technological Research Institute of University of Santiago de Compostela. In 2009 he joined the Computer Architecture Group of the Department of Electronics and Computer Science in the same University. He recently joined the Visualization and Data Analysis Department at Zuse-Institut Berlin. The scope of his research is centered in high performance computing using graphics processors.*

***Francisco Argüello Pedreira,*** *received the B.S. and Ph.D. degrees in Physics from the University of Santiago, Spain in 1988 and 1992, respectively. He is currently an Associate Professor in the Department of Electronic and Computer Engineering at the University of Santiago de Compostela, Spain. His current research interests include signal and image processing, computer graphics, parallel and distributed computing, and quantum computing.*

***Dora Blanco Heras****, received a M.S. degree in Physics in 1994 and a Ph.D. in 2000 from the University of Santiago de Compostela (Spain). She is currently an Associate Professor in the Department of Electronics and Computer Engineering at the same University. Her research is in the field of parallel and distributed computing, including, for example, software optimization techniques for emerging architectures, and on computer graphics and image processing.*