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

Efficient Acquisition and Clustering of Local Histograms for Representing Voxel Neighborhoods

Christian Meß and Timo Ropinski

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

Abstract

In the past years many interactive volume rendering techniques have been proposed, which exploit the neighboring environment of a voxel during rendering. In general on-the-fly acquisition of this environment is infeasible due to the high amount of data to be taken into account. To bypass this problem we propose a GPU preprocessing pipeline which allows to acquire and compress the neighborhood information for each voxel. Therefore, we represent the environment around each voxel by generating a local histogram (LH) of the surrounding voxel densities. By performing a vector quantization (VQ), the high number of LHs is than reduced to a few hundred cluster centroids, which are accessed through an index volume. To accelerate the required computational expensive processing steps, we take advantage of the highly parallel nature of this task and realize it using CUDA. For the LH compression we use an optimized hybrid CPU/GPU implementation of the k-means VQ algorithm. While the assignment of each LH to its nearest centroid is done on the GPU using CUDA, centroid recalculation after each iteration is done on the CPU. Our results demonstrate the applicability of the precomputed data, while the performance is increased by a factor of about 10 compared to previous approaches.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

In many application areas of volume rendering, knowledge regarding the neighborhood of a voxel is necessary during rendering. For instance, during a medical diagnosis or preoperative planning, it can be essential to know which structures are adjacent [MNTP07]. Another example is the application of advanced illumination models, where knowledge of a voxel's neighborhood is essential in order to compute appropriate shading effects. Among these techniques are occlusion-based shading effects [Ste03, RMD*08] which are known to improve the spatial comprehension [TCM06] by incorporating the neighborhood of a voxel during rendering. Additionally, Lundström et al. could show, that the neighborhood information of a voxel can be exploited in order to design meaningful transfer functions [LLY05, LLY06]. While all mentioned techniques are tailored towards interactive volume rendering, the acquisition of a voxel's neighborhood during rendering time, would have a serious impact on the rendering speed and hence not permit interactive frame rates. When for example considering a neighborhood having the radius $r = 24$, the information of already $\frac{4}{3}\pi r^3 \approx 57.900$ voxels has to be taken into account. Since getting the information for each voxel during rendering would require an additional 3D texture fetch, this cannot be done on-the-fly even with current graphics processing units. Therefore, often a preprocessing is exploited in order to generate the desired neighborhood information [Ste03, RMD*08]. While preprocessing solves the problem of expensive neighborhood data acquisition during rendering, the vast amount of generated data does still not allow interactive volume rendering. Therefore, researchers have proposed techniques, which allow an efficient representation of the local neighborhood information. Local histograms (LHs) are such a representation, they have been already exploited in the area of volume graphics [LLY05, LLY06, RMD*08]. While LHs represent the distribution of values in a specified area, they do not contain any spatial information. It has been shown, that several application examples, as transfer function design as

DOI: 10.2312/VG/VG10/117-124

well as ambient occlusion can disclaim the spatial information by still achieving reasonable results. However, storing a LH for each voxel still requires an enormous amount of memory. When for instance dealing with a data set stored with a precision of 12 bits, each LH would already consume $4096 \times 2Byte \approx 8KByte$, whereby 4096 is the number of required bins and we assume that all occurrences of one intensity value can be represented by 2 Bytes. For a data set having $512 \times 512 \times 512$ voxels, this would result in 1 terra byte of additional data. Obviously, dealing with this amount of additional data during rendering is not feasible. Therefore, LH compression can be exploited, whereby a promising way to reduce the number of LHs is to apply clustering techniques. This is also the direction, which we follow in this paper. By exploiting the parallel nature of the generation and the clustering of LHs, we are able to reduce the amount of histograms accessed during rendering to a few hundred cluster centroids. These centroids are accessed during rendering through an index volume. We have chosen to exploit a hybrid CPU/GPU implementation of the k-means vector quantization (VQ) algorithm. Since the clustering algorithms operate in a vector space of high dimensionality, namely the number of bins of the LHs, i. e., 4096 for 12 bit data, the clustering is a very time consuming process. In this paper we will show, how the CUDA architecture can be exploited in order to accelerate LH generation and clustering by a factor of about 10 as compared to previous approaches. We will explain how to exploit the shared memory provided by current GPUs in order to allow data caching and thus improve the computation speed. Thus the required computation time becomes manageable, and the door is opened for other applications of LH techniques.

In the remainder of this paper we will first discuss some related work in Section 2, before describing our hybrid GPU/CPU processing pipeline in Section 3. Furthermore, we will discuss performance results and the influence of parameters on the output quality. To demonstrate the applicability of the generated data, some application examples are discussed in Section 5. Finally, the paper concludes in Section 6.

2. Related Work

In recent years, several interactive volume rendering techniques have been proposed, which exploit neighboring information in order to improve the visual results. In the following we briefly review the literature regarding two main areas, where neighboring information is used: transfer function design and advanced illumination. Lundström et al. where the first, who have employed a voxel's neighborhood in order to support the transfer function design process [LLY05, LLY06]. By computing LHs, more specifically so-called Partial Range Histograms, they were able to classify and visually separate structures, which where not classifiable with conventional transfer function approaches. To incorporate the information provided by the LHs, they showed how to use them as an additional dimension within the transfer function design process. Thus, tissues overlapping in intensity space could be visually separated. A similar though different approach has been proposed by Patel et al. [PHBG09]. They define a transfer function based on the first and second statistical moments. While in the approach by Lundström et al. [LLY05] the neighborhood for which the LH has been precomputed is fixed, Patel et al. derive the statistical properties based on a growing neighborhood centered at each voxel. By detecting trends in these values, they were able to provide a 2D projection, which is facilitated in the transfer function design process. Haidacher et al. have extended this concept and proposed a more intuitive user interface for statistical transfer functions, and show how the incorporated properties could still be exploited when the data set is subject to noise [HPB*10]. Correa et al. have proposed a technique, which nicely demonstrates the dependencies between transfer function design and advanced illumination [CM09]. They provide a technique, which can be used to classify data directly based on the ambient occlusion of a voxel. In their paper they present how to derive so-called occlusion patterns during a preprocessing and present them within the occlusion spectrum, which is used as the foundation of their transfer function space. While all discussed techniques have been tailored at transfer function design, occlusion based shading has also been enabled by exploiting precomputed voxel neighborhoods. Stewart has presented a technique called vicinity shading [Ste03]. With vicinity shading it becomes possible to simulate illumination of isosurfaces within a volumetric data set. Therefore, in a precomputation the vicinity of each voxel is analyzed and the resulting value, which represents the occlusion of the voxel, is stored in a shading texture which can be accessed during rendering. Ropinski et al. have exploited a similar approach [RMD*08]. Based on LHs, they determine the occlusion of a voxel. By modulating the precomputed LHs with the transfer function during rendering, they are able to achieve occlusion-based shading effects during rendering interactively. Thus, ambient occlusion as well as color bleeding can be simulated. However, their technique requires an expensive preprocessing algorithm in order to generate the neighborhood information, which is independent from the transfer function.

To reduce the amount of data produced by the mentioned preprocessing techniques, k-means clustering can be used. In the last years a number of approaches to accelerate k-means by mapping parts of the algorithm onto the GPU were introduced. Che et al. describe how to map general-purpose applications on GPUs using CUDA [CBM*08]. Among other applications they focus on data mining and show a CUDA implementation of k-means. To speed up data access, all centroids are stored in constant memory. The CUDA implementation achieves a 72× speedup compared to a single-threaded and a 35× speedup compared to a four-threaded

CPU implementation. Unfortunately this technique is not feasible for our purposes, because on recent CUDA capable GPUs only 64 KB of constant memory is available. Our typical centroid dimension is 256 and we are dealing with floating point numbers. So, in our application, constant memory would only be sufficient for 64 centroids. Zechner and Granitzer [ZG09] present a hybrid approach to accelerate k-means clustering. The labeling stage is performed on the GPU using CUDA, whereas the centroid recalculation is done completely on the CPU. To measure speedup they compare computation times for various dimensions, data set sizes and code book sizes with CPU based implementations. In comparison to a fully SIMD optimized CPU implementation a 14× speed improvement could be shown. While our approach is inspired by their proceeding, we have extended it and integrated a bricking approach, to allow to further exploit parallelism and deal with data, which does not fit into GPU memory. Wu et al. focus on clustering of very large data sets and propose a stream-based algorithm [WZH09]. CUDA supports asynchronous memory transfers and streaming, which allows GPU and CPU computations to overlap with memory transfers. Data sets not fitting into GPU memory are partitioned into blocks which are distributed among the available streams. Compared to an unoptimized single core CPU implementation a 300× speedup and compared to a highly optimized CPU version running on eight cores a 29.3× speedup could be observed. In contrast to the preceding methods Hong et al. [HtLlDt*09] show how both steps, data object assignment and centroid recalculation, can be offloaded to the GPU. This is achieved by an intermediate step after each iteration. Cluster labels are downloaded to the CPU and two structures, a reordered label data set and a data set counting the data objects in each cluster, are computed. These structures are then uploaded to the GPU and used within centroid recalculation.

3. Local Histogram Generation and Clustering via CUDA

While all the techniques discussed in Section 2 exploit neighborhood information, there are mainly two issues. First, the precomputation times are very long, and second, in many cases a large amount of data is generated, which needs to be compressed in order to make it handy during rendering. While the first problem can be addressed by exploiting the parallel processing power of current GPU architectures, the second problem can be faced by applying sufficient clustering techniques. In this section we describe how to exploit the parallel processing power and realize the clustering using CUDA. Before explaining our approach, a short description of the CUDA hardware and programming model is given. The precomputation itself comprises of two main steps, which are later on covered in more detail. In the first preprocessing step a LH for each voxel is computed representing the distribution of voxel densities within a sphere with radius r. In the second step a clustering of these n LHs

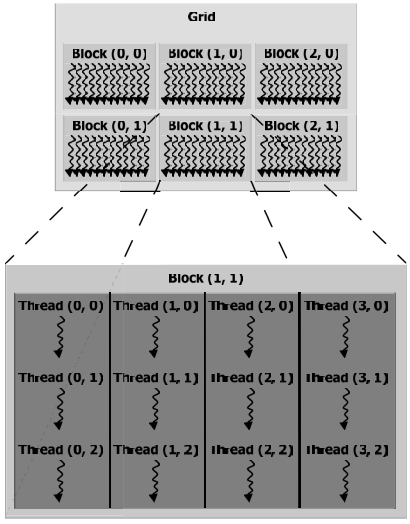

Figure 1: *CUDA thread hierarchy: Threads are organized in up to three dimensional blocks and blocks are grouped in a two dimensional grid (modified from [NVI09]).*

is performed using the k-means algorithm resulting. The outcome of this clustering are $m \ll n$ LHs, which are the centroids of the identified clusters. Thus we generate m LHs as well as a volumetric index data set, which relates each voxel of the original data set to one of the m histograms.

3.1. Compute Unified Device Architecture (CUDA)

In the CUDA programming model threads are extremely lightweight. Thread creation and scheduling is very inexpensive and so a low granularity decomposition of problems can be achieved. With respect to volume rendering, a typical design goal for a CUDA algorithm is that each thread handles the computation for a single voxel. To manage this huge amount of threads the CUDA programming model arranges threads in an hierarchy as shown in Figure 1. Threads are grouped in thread-blocks which are again organized in a grid. A grid is executed within a kernel on the GPU and thread-blocks are scheduled onto the processors by the CUDA runtime [NVI09].

In order to utilize the GPU resources in an efficient way it is crucial to optimally exploit the CUDA hardware model and especially the different memory types (see Figure 2). On a CUDA capable GPU a number of multiprocessors (MP) resides. Each MP consists of eight Scalar Processor (SP) cores and a small amount (16 KB) of fast on-chip read-write shared memory (SM), which can be shared among the SPs. For acceleration of texture fetches a texture cache is available and each thread can access the 64 KB of very fast read only constant memory. Device memory is the slowest type of memory, but readable and writable from both host (CPU) and device (GPU).

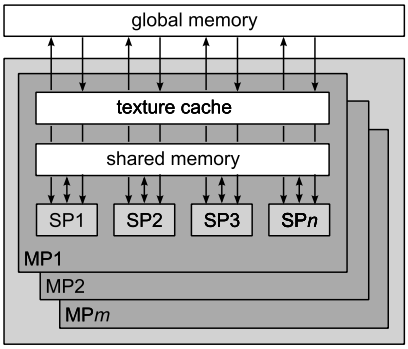

Figure 2: *CUDA hardware model: Arrows indicate the direction of memory access. White boxes indicate memory components and light gray boxes stand for processing units. Multiple scalar processors (SPs) are arranged within multi processors (MPs).*

3.2. Local Histogram Generation

Throughout the rest of this paper we refer to $LH(x)$ as the local histogram for voxel x and LH as the complete set of local histograms for the entire volume data set. The $LH(x)$ of a voxel x incorporates all voxels $\tilde{x}$ within a sphere $S_r(x)$ with radius r around the given voxel x. An $LH(x)$ consists of b bins, where b is the bit depth of the underlying volume data set. Each bin $LH_k(x), 0 \leq k < b$ sums up the amount of voxels with intensity k within $S_r(x)$. To incorporate distances, the contribution of a single voxel $y \in S_r(x)$ to a bin $LH_k(x)$ can be weighted by the distance between x and y. Because in many cases for classifying the neighborhood of x, only the relative distribution of $LH(x)$ is relevant, we normalize each $LH(x)$ such that $\sum_{k=1}^{b} LH_k(x) = 1$.

A workflow of our algorithm for GPU-based LH generation is shown in Figure 3. For checking if a voxel y lies in the sphere $S_r(x)$ with radius r and center x, and to compute weighted $LH(x)$ contributions, it is necessary to calculate the distance between x and y. This distance has to be computed approximately $\frac{3}{4}\pi r^3$ times, i. e., for every voxel in $S_r(x)$. When using the Euclidean distance

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$$

especially the square root calculation will slow down the $LH(x)$ generation, because this operation is a rather expensive CUDA function. To avoid this, we precompute a distance mask on the CPU and transfer the results to the constant memory of the GPU. The precomputed values are stored in a 3D array and thus the distance can be obtained by fetching the 3D array at the offset coordinates of y in relation to x. Because constant memory is a limited resource (64 KB for each SM), only distances for positive x, y and z values (one quarter of a hemisphere) are stored. The other distances are derived through sign changes, which can be performed at virtually no performance costs.

After uploading the distance lookup array to global device memory the volume data set is decomposed into bricks to support volume- and LH-sizes exceeding device RAM. This is necessary, since for a small volume data set with dimension 128^3 a LH of size 2 GB (128^3 * 256 bins * 4 byte) is calculated. Thus, special border treatment is needed for bricking in order to produce correct LHs for voxels whose spheres don't fit completely into a brick. Therefore, the bricks overlap each other by sphere radius r.

The CUDA kernel for LH generation is organized in the following way. Each thread is responsible for calculating the $LH(x)$ of one voxel x. The position of this voxel inside the volume brick depends on the thread- and block-indices of the thread. For each thread, space for one $LH(x)$ is allocated in SM. Because of the maximal block-size of a kernel call, the number of threads running on a MP at a time, is limited by the size of SM. On current GPUs the size of SM is 16 KB and a $LH(x)$ with 256 bins occupies 1 KB. Thus, 16 threads per thread-block is the upper limit, but the usable size of SM can be decreased by using many local variables within a kernel. In our implementation good results were achieved using a block size of $(2 \times 2 \times 2)$. In some situations a block size of $(2 \times 1 \times 2)$ seems to be even faster. With regard to the CUDA Hardware Model these low thread counts are suboptimal leading to a low utilization of the GPU. Future GPUs with an increased SM size will allow a much better utilization and significantly speed up our algorithm. To exploit data locality using the texture cache the volume bricks uploaded to device memory are bound to 3D-textures. The processing is performed by consecutively uploading the bricks to global

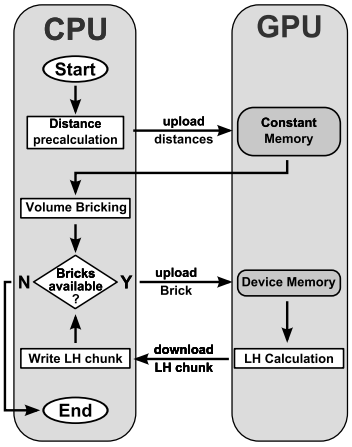

Figure 3: *GPU-supported LH generation: While the distance mask and the bricks are calculated on the CPU, the actual LH calculation as performed on the GPU. After each calculation step, the LH chunks containing the results are downloaded to the CPU.*

device memory for computing the LHs one per thread. During this computation, we iterate over all voxels within the neighborhood of radius r and write their contribution into the corresponding bin of LH within SM. When all voxels have been processed and thus the LH has been generated, every thread normalizes the computed histogram and writes it to device memory. After all bricks have been processed, they are transferred to the host system (the CPU in Figure 3) and written to file system. The algorithm stops when all bricks have been processed.

3.3. Local Histogram Clustering

With the technique described above a normalized local histogram $LH(x)$ was generated for each voxel x. In principle this data can be used already during rendering, but the size of LH will rapidly exceed device memory making interactive frame rates unfeasible. To solve this problem the amount of data has to be compressed. We use a clustering algorithm to achieve this goal. The result of the clustering is a smaller set of local histograms, the centroids of the clustering process, which we denote as a code book (cb), and a volumetric data set, which assigns each voxel to a cluster, the labeling volume.

As clustering method we have chosen the k-means clustering algorithm. k-means is a commonly used clustering algorithm initially proposed by MacQueen [Mac67]. In general k-means consists of two steps. Within the first step the data set is subdivided into k subsets by assigning each data point to its nearest centroid. When all data points are assigned, the centroids are recalculated in a second step by finding the center of each cluster. These steps are iterated until no data point moves from one cluster to another. To start the algorithm k centroids have to be chosen randomly or by using special seeding algorithms. Pena et al. give a comparison of various k-means initialization methods [PLL99]. Our GPU-based implementation is inspired by the approach presented by Zechner and Granitzer [ZG09], while we have integrated a bricked processing scheme, to support LHs and data sets exceeding GPU memory. The phases of the algorithm are outlined in Figure 4. In the first step a starting cb is generated on the CPU by randomly choosing n $LH(x)$ out of LH. In the following these n histograms are also denoted as the code vectors (cv) of the codebook (cb). All $cv_i \in cb$ with $0 \leq i < k$ are sorted by their mean intensity value in ascending order.

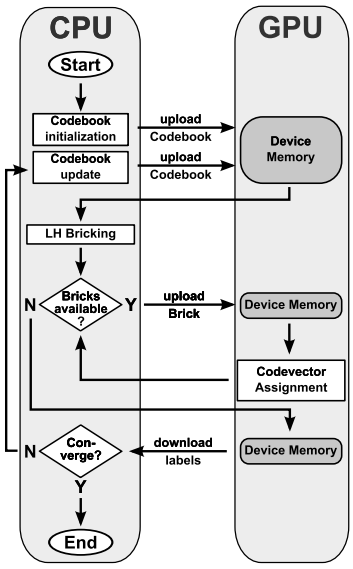

Figure 4: *GPU-supported LH clustering: In our hybrid approach, first codebooks are initialized on the CPU and uploaded to GPUs device memory. As long as bricks need to be processed, they are also uploaded sequentially and the codevector assignment is done until the algorithm terminate.*

The initial cb is transferred to device memory. In order to fit into device memory the LH has to be split in bricks. Each brick is uploaded to the GPU and for each $LH(x) \in LH$ the index of the $cv \in cb$ with the smallest distance to $LH(x)$ is determined. This is achieved by using the CUDA kernel shown in Listing 5, where each thread determines the minimal distance to one $LH(x)$. The $LH(x)$ on which the thread works is defined by the block- and thread-index. In detail each thread iterates over all $cv \in cb$ and stores the distance and the index of the cv with the minimal distance to the $LH(x)$ in a local variable. Because all threads in a block are operating on one cv in parallel, the cv is stored in fast SM. When all threads in a block have determined the index of the cv with minimal distance, this index is written to the appropriate position in the label data set. This label data set has the same dimension as the source data set. When all bricks are processed the labels are downloaded to the CPU and a new ordered code book cb_{new} is constructed. The algorithm converges to an approximately optimal cb if the distance $dist(cb_{new}, cb_{old})$ falls below a pre-defined threshold.

4. Performance Results

Table 1 shows the precomputation times achieved with our approach. All tests have been performed on an AMD Phenom 8750 Triple-Core processor system with 2400 MHz (4 GB RAM), equipped with a GeForce 9800 GTX+ GPU (512 MB RAM). The Cornell box results show the dependency of the precomputation time on the sphere radius r and number of codewords cw. The histogram generation took between 2.45 and 19.58 minutes for sphere radii between 8 and 24 voxels. During the clustering, the number and length of iterations heavily depends on the number of codewords. While for 256 codewords, one iteration takes only about 0.4 minutes, when dealing with 2048 codewords, $1.92 - 2.85$ minutes are used. Since the clustering is performed on normal-

data set	size (voxel^3)	sphere radius	hist. gen. (min.)	codewords	iterations	VQ (min.)	time per iteration (min.)	LH size	codebook size
Cornell box	$128 \times 128 \times 128$	12	2.45	256	6	2.37	0.40	2 GB	256 KB
Cornell box	$128 \times 128 \times 128$	8	1.48	2048	9	17.29	1.92	2 GB	2 MB
Cornell box	$128 \times 128 \times 128$	24	19.58	2048	19	54.19	2.85	2 GB	2 MB
Head	$192 \times 192 \times 110$	12	10.16	2048	11	59.07	5.37	3.9 GB	2 MB
Hand	$244 \times 124 \times 256$	24	78.28	2048	10	101.58	10.16	7.4 GB	2 MB
Feet	$128 \times 64 \times 128$	12	1.26	2048	12	9.17	0.76	1 GB	2 MB

Table 1: *CUDA accelerated preprocessing times for selected data sets. Preprocessing was performed on AMD Phenom 8750 Triple-Core processor at 2400 MHz (4 GB RAM) and GeForce 9800 GTX+ GPU (512 MB RAM).*

C:\Users\ropinski\Desktop\vq.c 1

```
float dist = 0.0f;
float minDist = CUDART_MAX_NORMAL_F;
int centroid = -1;

// for all codeVectors
for(int i=0; i<sizeCodeBook; ++i) {
    // load one cv component to SM
    if(tid < numBins) {
        cv[tid] = cb[i*numBins+tid];
    }
    __syncthreads();

    // calc distance
    dist = 0.0f;
    for(int j=0; j<numBins; ++j) {
        dist += pow(lh[gid]  - cv[j], 2);
    }

    if(dist < minDist) {
        minDist = dist;
        centroid = i;
    }
}
__syncthreads();

// write centroid index to device mem
dVolume[gid] = centroid;
__syncthreads();
```

Figure 5: *The main loop of the k-means algorithm as realized with our CUDA kernel.*

ized LHs, as described above, the radius r has only a minor influence on this stage, in that sense that larger r may result in more similar LHs. The Feet, Head and Hand data set have been preprocessed with exactly the same parameters as described by Ropinski et al. [RMD*08]. Thus, to provide some reference for our performance gain, we will compare our results to their results. However, it should be noted, that their technique is different in the sense, that the clustering is performed on packed histograms, usually having a dimension of 64. Our technique works on the original histograms potentially leading to better results for applications, where a higher accuracy is required. When applying our technique to packed histograms a further significant speedup could be gained. Although these differences exist, Ropinski et al.'s technique [RMD*08] is the most similar, and therefore chosen for comparison. As it can be seen, for the Feet data set ($r = 12$, $cw = 2048$), our histogram generation takes only 1.26 minutes compared to 15.98 minutes of the previous approach (speedup: 12.68). Furthermore, we could reduce the clustering time for this data set from 320.81 minutes down to 9.17 minutes (speedup: 34.98). A performance gain could also be achieved for the Head data set ($r = 12$, $cw = 2048$), where we require 10.16 minutes for the histogram generation as compared to the 16.63 minutes of the multi-core CPU implementation (speedup: 1.64). In this case the clustering could be improved to last only 59.07 as compared to 528.58 minutes (speedup: 8.95). In [RMD*08], the authors present also a timing of 61.60 minutes for the training phase of the clustering based on this data set. However, this has been achieved by simplifying the clustering (and also considering packed histograms), which potentially leads to less accurate results. For the Hand data set ($r = 12$, $cw = 2048$), we were able to reduce the histogram generation from 514.31 minutes to 78.28 minutes (speedup: 6.57), and the clustering from 633.80 minutes down to 101.58 minutes (speedup: 6.24). It should be noted, that the clustering times presented in this paper are for the entire clustering process, and not just for the VQ training as it is the case in [RMD*08].

Thus, dependent on the data set and preprocessing parameters, we are able to achieve a performance gain of $1.64 - 12.68$ for the histogram generation and of $6.24 - 34.98$ for the clustering. While these reported processing times make the environment generation for volumetric data sets already usable, we are confident that newer graphics hardware will result in an additional performance gain.

5. Application Examples

To demonstrate the applicability of the presented precomputation technique, we will present some application results within this section, which are based on the dynamic ambient occlusion technique presented by Ropinski et al. [RMD*08]. As mentioned in Section 2, Ropinski et al. have used LHs to simulate advanced illumination effects. By exploiting this transfer function independent representation, they are able to change all important rendering parameters interactively. To achieve this, they modulate the stored LHs with the transfer function during rendering, by multiplying each bin's value

$LH_j(x)$ with the opacity $\tau_\alpha(j)$ and the color $\tau_{rgb}(j)$, as assigned with the transfer function to intensities equal to j: $c_j = \tau_\alpha(j) \cdot \tau_{rgb}(j) \cdot LH_j(x)$. The result of the thus modulated bin values are composited and weighted by the number of voxels represented by the LH. Hence, for each LH an environmental color c'_j can be obtained. To just compute an approximation of the ambient occlusion, $\tau_{rgb}(j)$ can be neglected from the previous equation.

During rendering the computed c'_j can be assigned to each voxel by using the index volume storing the cluster IDs. This requires only one additional 3D texture fetch and thus makes the environmental color available with comparable little overhead. We have tested our preprocessed volumes with this simple rendering technique. The results are shown in Figure 6, whereby the subfigures (a) and (b) have been generated with our technique and (c) shows a comparison taken form the original dynamic ambient occlusion paper [RMD*08]. Both used data sets have been preprocessed with a sphere radius of $r = 12$, while for the Cornell box data set 256 codewords and for the Visible Human head data set 2048 codewords have been used. As it can be seen, the visual quality of the images is comparable to the original ones, although significant less preprocessing time is required.

While the presented images show the applicability of the data for dynamic ambient occlusion shading, we believe, that other approaches could also benefit from the improved precomputation times. Probably the easiest integration would be to exploit our data for the transfer function design based on LHs, as described by Lundström et al. [LLY06].

6. Conclusions and Future Work

In this paper we have presented a hybrid GPU/CPU technique for the generation and clustering of LHs in volumetric data sets. By exploiting the capabilities of modern stream processing APIs, we are able to perform this preprocessing about 10 times faster, than previous reported approaches [RMD*08]. This performance gain is achieved by exploiting the shared memory of current GPUs, which is accessed by our CUDA implementation. Furthermore, by performing some subtask of the used k-means VQ algorithm on the CPU and others on the GPU, the strengths of both units can be combined. Thus, while the assignment of each LH to its nearest centroid is done on the GPU using CUDA, centroid recalculation after each iteration of the VQ is done on the CPU. We have demonstrated the applicability of the generated histograms by showing dynamic ambient occlusion examples as well as histogram-dependent transfer functions. Both are promising techniques, which have been proposed before, but are so far not widely used. One reason for that might be the lack of efficient LH computation methods. We believe, that the reported preprocessing times are manageable and this LH generation based on volumetric data sets becomes practical.

In the future, we would like to provide our computation algorithm as an open source library for other researchers. Thus, the functionality can be adapted and new techniques based on LHs can be developed. Furthermore, we would like to offload codebook recalculation on to the GPU, as demonstrated by Hong et al. [HtLlDt*09]. Besides this optimization, we see further potential performance improvements techniques. For example, we could adapt the histogram packing technique proposed by Ropinski et al. [RMD*08]. We believe, that besides the discussed approaches, other examples could also benefit from the usage of LHs. For instance, to allow an interactive contrast enhancement based on LHs, our approach could be helpful.

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).

References

[CBM*08] CHE S., BOYER M., MENG J., TARJAN D., SHEAFFER J. W., SKADRON K.: A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput. 68*, 10 (2008), 1370–1380. 2

[CM09] CORREA C., MA K.-L.: The occlusion spectrum for volume classification and visualization. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1465–1472. 2

[HPB*10] HAIDACHER M., PATEL D., BRUCKNER S., KANITSAR A., GRÖLLER M. E.: Volume visualization based on statistical transfer-function spaces. In *Proceedings of the IEEE Pacific Visualization Symposium 2010* (2010), p. to appear. 2

[HtLlDt*09] HONG-TAO B., LI-LI H., DAN-TONG O., ZHAN-SHAN L., HE L.: K-Means on commodity GPUs with CUDA. In *Computer Science and Information Engineering, World Congress on* (Los Alamitos, CA, USA, 2009), vol. 3, IEEE Computer Society, pp. 651–655. 3, 7

[LLY05] LUNDSTRÖM C., LJUNG P., YNNERMAN A.: Extending and simplifying transfer function design in medical volume rendering using local histograms. In *EuroVis* (2005), pp. 263–270. 1, 2

[LLY06] LUNDSTRÖM C., LJUNG P., YNNERMAN A.: Local histograms for design of transfer functions in direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 6 (2006), 1570–1579. 1, 2, 7

[Mac67] MACQUEEN J. B.: Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability* (1967), Cam L. M. L., Neyman J., (Eds.), vol. 1, pp. 281–297. 5

[MNTP07] MÜHLER K., NEUGEBAUER M., TIETJEN C., PREIM B.: Viewpoint selection for intervention planning. In *EuroVis* (2007), pp. 267–274. 1

[NVI09] NVIDIA: CUDA programming guide 2.3, 2009. 3

[PHBG09] PATEL D., HAIDACHER M., BALABANIAN J.-P., GRÖLLER M. E.: Moment curves. In *Proceedings of the IEEE Pacific Visualization Symposium 2009* (2009), pp. 201–208. 2

(a) (b) (c)

Figure 6: *The preprocessed data sets have been tested by applying dynamic ambient occlusion and color bleeding [RMD*08]. In (a) only the ambient occlusion approximation is applied, while in (b) also the environmental color has been exploited. (c) shows a comparison to (b) taken from the original paper on dynamic ambient occlusion [RMD*08].*

[PLL99] PEÃŚA J., LOZANO J., LARRAÃŚAGA P.: An empirical comparison of four initialization methods for the k-means algorithm. *Pattern Recognition Letters 20*, 10 (1999), 1027 – 1040. 5

[RMD*08] ROPINSKI T., MEYER-SPRADOW J., DIEPENBROCK S., MENSMANN J., HINRICHS K.: Interactive volume rendering with dynamic ambient occlusion and color bleeding. *Computer Graphics Forum 27*, 2 (2008), 567–576. 1, 2, 6, 7, 8

[Ste03] STEWART A. J.: Vicinity shading for enhanced perception of volumetric data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), p. 47. 1, 2

[TCM06] TARINI M., CIGNONI P., MONTANI C.: Ambient occlusion and edge cueing to enhance real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 6 (2006). 1

[WZH09] WU R., ZHANG B., HSU M.: Clustering billions of data points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop* (Ischia, Italy, 2009), ACM, pp. 1–6. 3

[ZG09] ZECHNER M., GRANITZER M.: Accelerating K-Means on the graphics processor via CUDA. In *Intensive Applications and Services, International Conference on* (Los Alamitos, CA, USA, 2009), IEEE Computer Society, pp. 7–15. 3, 5