Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

Load-Balanced Multi-GPU Ambient Occlusion for Direct Volume Rendering

A. Ancel [1 †] and J.-M. Dischler [2 ‡] and C. Mongenet [2 §]

[1] Inria – Grenoble, France
[2] LSIIT – University of Strasbourg – Strasbourg, France

Abstract

Ambient occlusion techniques were introduced to improve data comprehension by bringing soft fading shadows to the visualization of 3D datasets. They consist in attenuating light by considering the occlusion resulting from the presence of neighboring structures. Nevertheless they often come with an important precomputation cost, which prevents their use in interactive applications based on transfer function editing. This paper explores parallel solutions to reach interactive framerates with the use of a multi-GPU setup. Our method distributes the data to the different devices for computation. We use bricking and load balancing to optimize computation time. We also introduce two repartition schemes: a static one, which divides the dataset into as many blocks as there are GPUs and a dynamic one, which divides the dataset into smaller blocks and distributes them using a producer-consumer way. Results, using an 8-GPU architecture, show that we manage to get important speedups compared to a mono-GPU setup.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors—I.3.1 [Computer Graphics]: Parallel processing—I.3.3 [Computer Graphics]: Display algorithms—

1. Introduction

Shading is an important technique improving data comprehension when rendering volume datasets using Direct Volume Rendering (DVR), where there are multiple more or less transparent and interleaved layers. It helps the human eye perceiving the relative position and shapes of the different layers extracted by classification using the transfer function. In this context, one can use either direct illumination techniques like Phong Shading [Pho75], or indirect illumination techniques, that are usually more computationally expensive. Ambient Occlusion techniques fall into this second category. They bring comprehensive cues to volume rendering by providing soft and smoothly varying shadows, which emphasize the shape of structures in the volume. Figure 1 illustrates this difference by comparing Phong lighting only and Phong lighting combined with Ambient Occlusion (see improved nose bone cavity perception). Previous works have introduced this method as an additional precomputation step that produces a volume containing light attenuation factors. This volume is then used during the visualization step to modulate the color of the corresponding voxels. The main problem with this technique is that it remains expensive in terms of precomputation time. In addition, a new Ambient Occlusion volume must be recomputed whenever the transfer function is modified. Once it is precomputed, final rendering is as fast as classical DVR. Precomputation-free screen-space approximations merge occlusion computations with the rendering step like Schott et al. [SPH*09] and Šoltészová et al. [ŠPBV10]. But they affect the final rendering performance and only take into account a partial cone of data.

To preserve final rendering framerates, we propose to keep the computation of Ambient Occlusion as a precomputation step. Our contribution is to speed it up using a parallel implementation. We focus on multi-GPU architectures as they become widely spread. They are a cheaper alternative to clusters and provide parallel computational capabilities with the help of specific programming languages like CUDA or OpenCL. Nevertheless, data transfers between the

† alexandre.ancel@imag.fr
‡ dischler@unistra.fr
§ mongenet@unistra.fr

© The Eurographics Association 2012.

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

DOI: 10.2312/EGPGV/EGPGV12/099-108

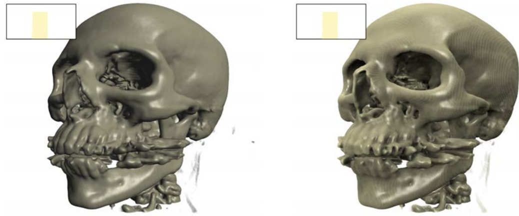

Figure 1: *Comparison of DVR with Phong shading without (left) and with (right) additional ambient occlusion.*

GPUs and CPUs have to be cautiously taken into account, as they might affect the performance of the whole process. To parallelize Ambient Occlusion computation on such a setup, we propose to distribute this computation to multiple GPUs. This raises the following issues: how do we scatter the data on the different devices, how do we ensure a good balance of the load and how do we manage to minimize data transfers between the different GPUs. Our main focus here is to provide solutions for these issues. We also show that this parallelization is compatible with a sort-last multi-GPU rendering pipeline, which has already been addressed in previous works.

This paper is organized as follows. Section 2 presents related works in the areas of Ambient Occlusion and multi-GPU techniques both applied to the context of direct volume rendering (DVR). In Section 3, we present the Ambient Occlusion formulation on which is based our technique in DVR. Section 4 introduces a bricking technique allowing to reduce the precomputation times. Section 5 then introduces our multi-GPU parallelization technique for Ambient Occlusion. Section 6 presents results and performance discussions. Finally, Section 7 concludes and discusses future works.

2. Related works

2.1. Ambient Occlusion for DVR

Light contributions from surrounding features bring important cues to the human eye as experimentally tested by Langer et al. [LB00]. Ambient Occlusion was introduced by Zhukov et al. [ZIK98] as the notion of obscurance to bring ambient illumination to objects without the cost of global illumination. As opposed to classical shading or shadowing methods, this technique is independent of lights and viewpoint. Stewart [Ste03] is the first to work on Ambient Occlusion applied to DVR. In his method called "vicinity shading", he proposes to precompute visibility factors based on voxel densities. Ruiz et al. [RBV*08] extend Stewart's work to the obscurance model, in which the distance to the occluder is used. To reduce computations, Hernell et al. [HLY07] compute Ambient Occlusion by considering only local spheres around voxels. They throw rays along which they accumulate opacities. The factors are stored in a 3D texture used in a final rendering pass to modulate colors. They further extend their previous works in [HLY10], by modifying the opacity of the transfer function to increase/decrease the shadowing effect. Ritschel [Rit07] bases his work on spherical harmonics to compute visibility in the volume dataset. With a precomputation time reduced by using the GPU, he produces soft shadows and attenuation. Desgranges et al. [DE07] try to evaluate the most frequent distances between so called "full" voxels through histogram computation. They then compute occlusion volumes based on a summed area volume. The final occlusion volume is obtained by blurring previously obtained occlusion volumes. Rezk Salama [Sal07] uses Ambient Occlusion and subsurface scattering in the context of isosurface rendering to produce high-quality renderings. Ropinski et al. [RMSD*08] compute the interaction of neighboring voxels using histograms, built independently of any transfer function. They combine a reduced number of histograms and a transfer function in the rendering pass to interactively render Ambient Occlusion and color bleeding effects. Meß and Ropinski [MR10] further improve the previous work and its precomputation cost by porting part of the technique on the GPU using CUDA. Schott et al. [SPH*09] propose a directional occlusion method, where light and view positions coincide. As it is an image space approach, no precomputations are required. Šoltészová et al. [ŠPBV10] further improve this technique by allowing the free placement of the light source. Correa and Ma [CM09] propose to use Ambient Occlusion to help with the classification. Ancel et al. [ADM10] propose to reduce the darkening introduced by Ambient Occlusion by separating the different layers to visualize and by computing Ambient Occlusion separately for each layer. Ruiz et al. [RSKU*10] compute Ambient Occlusion using the bounding box of the tangent sphere to the considered voxel by evaluating the portion that in unoccluded.

2.2. Multi-GPU

The use of multi-GPU setups for computation is a recent area of research, as the increasing programmability of graphic cards and the introduction of CUDA and OpenCL ease the use of such configurations. In the volume rendering field, several research areas have investigated the use of multi-GPU.

Direct volume rendering can benefit from multi-GPU configurations, as they allow the visualization of big datasets that do not fit in the memory of a single GPU. Marchesin et al. [MMD08] propose an experimental study comparing multi-GPU and cluster for sort-last volume rendering using big datasets. More recently, Fogal et al. [FCS*10] propose a method for visualizing large datasets using a cluster of multi-GPU computers. In the context of isosurface rendering, Martin et al. [MSM10] also use multi-GPU clusters to generate isosurfaces.

© The Eurographics Association 2012.

Regarding to the specific area of Ambient Occlusion and to our knowledge, no work has been conducted on parallelizing the computation of Ambient Occlusion on a multi-GPU setup.

3. Ambient Occlusion

Theory – In classic triangle-based rendering, Ambient Occlusion computation is based on the analysis of objects neighboring the considered point. This is represented as the integration of a visibility function V in the hemisphere described by the solid angle ω centered on normal N at considered point p:

$$AO_p = \frac{1}{\pi} \int_{\Omega} V_{p,\omega} (N \cdot \omega) d\omega \tag{1}$$

Ambient Occlusion in DVR is accurately computed by raycasting to sample the neighborhood as in the previous works conducted by Hernell et al. [HLY07]. They thus define the light arriving at the center x_v of voxel v from direction l by:

$$I_l(x_v) = \int_a^{R_\Omega} \frac{1}{R_\Omega - a} \cdot \exp\left(-\int_a^s \tau(u)\,du\right) ds \tag{2}$$

with a an initial offset, which allows one to lower the influence of nearest neighbors, R_Ω the considered sphere radius and τ the optical depth. This equation defines an occlusion region which is a shell of thickness $R_\Omega - a$ centered on each voxel. It is approximated using M samples for a front-to-back compositing scheme with an opacity α_i, obtained via the transfer function:

$$I_l(x_v) = \sum_{m=0}^{M} \frac{1}{M} \prod_{i=0}^{m-1} (1 - \alpha_i) \tag{3}$$

As this formula implies the Ambient Occlusion factors depend on the classification done with the transfer function, these factors need to be recomputed with each change of the function.

Obtaining the Ambient Occlusion factor for voxel v is finally a matter of raycasting and combining the contributions computed per-ray for a set of L rays in the following way:

$$AO(v) = \frac{1}{L} \sum_{l}^{L} I_l(x_v) \tag{4}$$

Implementation – Our implementation of Ambient Occlusion relies on CUDA. We divide the dataset into 2D CUDA blocks virtually mapping a part of each slice in Z direction. The CUDA grid is then built by putting in the first dimension the number of blocks in a Z-slice and in the second one the number of slices to process.

At launch and for a given ray direction, each kernel then processes a voxel, computes its Ambient Occlusion factor for the current ray direction and sums it to a float accumulation buffer. Each value of this texture will then finally be divided by the number of rays and at the same time resampled to 8 bits for storage means. This method is based on the OpenGL method used by Hernell et al. [HLY07]. The kernel function is very simple as it only consists in a loop fetching opacity values in a texture using post-classification along a ray direction. In terms of optimization, it does not reuse data and does not use global memory, so it cannot be further optimized on the code side.

4. Bricking

To decrease Ambient Occlusion computation times, an efficient solution consists in skipping empty areas caused by classification. Indeed, when rendering the volume dataset, there is often some part of the voxels that are made invisible by the transfer function and thus do not contribute to the visualization. The aim of bricking is to avoid computing Ambient Occlusion values for these voxels. We do this by bricking the dataset into equal-sized cubic bricks. Each brick is marked with a binary information: either 1 for a *full brick*, i.e. a brick in which we will compute Ambient Occlusion factors for each voxel it contains, or 0 for an *empty brick* that does not contribute to the final rendering and thus can be skipped. As for Ambient Occlusion factors, the content of a brick depends on the transfer function, so it must be recomputed each time the latter is modified.

Computing brick information – The binary information associated with each brick is computed on the GPU by iterating over the voxels contained in a brick until a non transparent voxel is found (in this case the brick value is 1) or all the voxels in the brick are processed (in this case the brick value is 0). Each GPU receives from the CPU the same amount of bricks to process by subdividing the dataset into n equal-sized blocks B_i (see top of figure 2). The result of the computation is a binary texture containing either 1 for *a full brick* or 0 for *an empty brick*. Since the processing of each block is the same, no explicit load balancing is necessary in this case. In addition, as it will be shown in the results section, this computation is very fast and represents only a fraction of the computation time of Ambient Occlusion. To be efficient, bricking is also computed in a parallel way by processing equal-sized blocks on each GPU.

Artifacts prevention – The creation of bricks has to be done cautiously as we might have empty bricks neighboring full bricks and subsequently a voxel that needs an Ambient Occlusion factor neighboring a voxel that needs no factor. During the final rendering step, we are using trilinear interpolation, so we need to compute an Ambient Occlusion

© The Eurographics Association 2012.

value for voxels contained in empty neighboring bricks, to prevent visual artifacts in the form of shading discontinuities. To solve this problem, we use an overlap for each brick, so that we take into account neighboring voxels. We have experimented different brick sizes, as described in section 6.

The bricks are then used in the Ambient Occlusion precomputation pass. When a voxel is processed, we fetch the corresponding brick value: if the value indicates a full brick, we compute an Ambient Occlusion factor for this voxel, otherwise we exit the kernel. This results in performance improvement for the precomputation time.

5. Parallelizing Ambient Occlusion computation

Since Ambient Occlusion is computationally expensive, and therefore prevents interactive transfer function editing, we parallelize this computation on a multi-GPU architecture. We consider an architecture composed of n GPUs. In the following, we first discuss the general parallelization strategy that we apply in section 5.1. We then introduce two computation distribution strategies: static in section 5.2 and dynamic in section 5.3, while discussing their respective benefits and drawbacks.

5.1. Choosing the right loop

When analyzing the Ambient Occlusion computation of formula 4, we see that there are several loop levels involved as described in Algorithm 1.

Algorithm 1 Algorithm describing the ray casting process associated with Ambient Occlusion computation.

```
CPU: For each ray direction
GPU:   For each voxel in the dataset
GPU:     For each sample taken
GPU:       Accumulate opacity along the ray
```

The different loops process respectively 1) the ray directions to sample the neighborhood, 2) the voxels to obtain an Ambient Occlusion value for each and 3) samples along the ray. The third loop is not easily parallelizable as there is an order dependency between the samples taken along the ray. So, there are mainly two possibilities: parallelization according to ray directions or according to voxels.

The first possibility consists in parallelizing the computation according to the ray directions. To do so, we can split the set of ray directions into n subsets, n being the number of available GPUs on the computer. Each GPU computes the Ambient Occlusion value corresponding to the sphere sampling induced by its ray subset. We then have to merge the different Ambient Occlusion values computed into a single value according to formula 4. This can be done either in a software manner, by transferring and merging every Ambient Occlusion volume on the CPU, or in a hardware manner, where one GPU would be in charge of merging sub-volumes that the other GPUs would send. In either way, this method implies prohibitive data transfers.

The second possibility is more interesting as it matches a natural sort-last parallel rendering pipeline. It consists in splitting the dataset into as many blocks as there are GPUs. The blocks are chosen to be parallelepipeds in order to keep all subsequent computations efficient in terms of data locality and cache usage. Each block is transfered to the associated GPU, where the corresponding Ambient Occlusion volume is computed. This volume does not need to be transfered back, since the same GPU can then directly perform the volume rendering of this block. Only the corresponding image must be transfered to a master GPU for final image compositing. Nevertheless, one has to pay attention concerning the way blocks are defined. Indeed, Ambient Occlusion computation on one single voxel involves taking into account a large part of neighboring voxels because of R_Ω (see formula 2). This neighborhood must be taken into account when splitting the block. To do so, we add an overlap to the defined blocks in every direction. It corresponds to the number of voxels along the radius R_Ω of the sampling sphere.

We note that an additional parallelization approach could be used according to the work of Ancel et al. [ADM10], in which they propose to compute Ambient Occlusion factors based on a feature separation of the dataset used to improve the perception of different structures. This work adds an additional loop on features before iterating over the ray directions. This loop can be parallelized by computing Ambient Occlusion values for each feature on a different GPU. However, when combined with bricking, processing features in parallel might result in severe unbalanced load, because the sizes of the features are generally uneven, thus resulting in different computation requirements.

5.2. Static distribution

In this section, we describe a method for statically distributing Ambient Occlusion factors computation. In the following, we discuss three main aspects : data distribution, load-balancing and rendering.

5.2.1. Data distribution

With a static distribution the dataset is decomposed into n blocks (for n GPUs), each block being assigned to one GPU, which computes all Ambient Occlusion factors related to this block. If the block is larger than the corresponding available GPU memory, it is further divided into sub-blocks that are then processed sequentially within the GPU.

A naive approach consists in splitting the dataset into equal sized blocks B_i with $i \in [1,n]$. All voxels of block B_i are then scanned and Ambient Occlusion values computed according to formula 4. Once this computation is completed,

© The Eurographics Association 2012.

we can render the block and send back the corresponding image for compositing. The parallel computation of Ambient Occlusion factors is possible, since they are independent from each other. But this approach is not efficient, since it does not exploit empty spaces. The bricking technique introduced in section 4 allows us to take into account empty spaces to improve global performance. Empty spaces introduce unbalance in the computation times since corresponding empty bricks might not be equally distributed among the different blocks B_i. We propose a technique to address this issue by choosing a load-balancing criterion. In the next sections, we respectively describe how bricking is performed and how the load balancing issue is addressed.

5.2.2. Addressing the load balancing issue

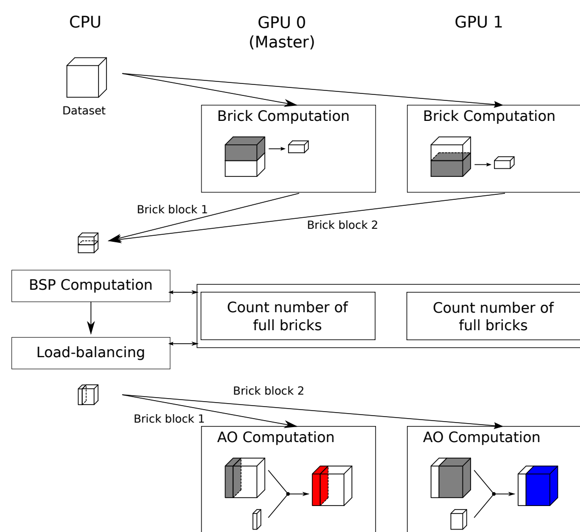

Figure 2: *Load-balancing process for computing Ambient Occlusion.*

Depending on the classification resulting from the transfer function, one can get large empty areas in the dataset, resulting in empty bricks that are skipped during Ambient Occlusion computation. The distribution of empty bricks depends on both the dataset and the transfer function. Subdividing the dataset into equal-sized blocks results in unbalanced workload, since some blocks are likely to contain more empty bricks than others. To improve load-balancing, we have to improve the block distribution on the GPUs, in a way such that the Ambient Occlusion computation would take approximately the same time on each GPU.

To perform load balancing, we first need a criterion that allows us to evaluate the computation cost of a given block. This criterion is linked to the number of full bricks inside this block. We thus need to count the number of full bricks in a given block. This task can be done in parallel by the different GPUs. Once we have set up a criterion, we can use it to compute a subdivision into blocks resulting in similar workload for each GPU. Section 5.2.3 describes our load-balancing criterion and section 5.2.4 shows how to compute the number of full bricks required by this criterion.

The load balancing method that we have implemented works as follows. In a first step, we split the dataset into adjacent blocks using a BSP-tree. In a second step, we move the splitting plane between blocks according to the criterion. These two steps are respectively described in section 5.2.5 and section 5.2.6.

5.2.3. Load-balancing criterion

Let us assume that block B_i contains n^i_f full bricks and n^i_e empty bricks. The time to compute an empty brick is denoted by T_e and the one to compute Ambient Occlusion factors on a full brick is denoted by T_f. T_e is almost completely negligible compared to T_f since one only has to check whether the brick is empty or not, whereas T_f is a constant depending on the size of the brick and on the radius R_Ω. We thus propose to use as single criterion to evaluate the cost for block B_i: n^i_f, i.e. the number of full bricks in B_i. We then can evaluate the balancedness between two different blocks B_i and B_j ($i \neq j$) by computing $D(B_i,B_j) = [n^i_f] - [n^j_f]$. The aim of global load balancing is to bring the result of this formula towards 0 for all pairs (B_i,B_j) with $(i,j) \in [1,n]^2$ and $i \neq j$. Formally, this can be expressed as finding out the set of blocks $S = \{B_k | k \in [1,n]\}$ among $\wp$, the set of all possible partitions of the dataset into n blocks, such that it minimizes the maximum value of $D(B_i,B_j)$ for all pairs of blocks:

$$S = \{B_k \mid k \in [1,n]\} \in \wp \mid M_S = min\left\{M_{S_q} \mid S_q \in \wp\right\} \quad (5)$$

$$M_{S_q} = max\left\{|D(B_i,B_j)| \mid \forall (B_i,B_j) \in S^2_q,\ i \neq j\right\} \quad (6)$$

This equation is a mathematical translation of the fact that all blocks should have roughly the same amount of full bricks. Load balancing is a matter of determining the optimal set S according to this equation. The problem is that there are many possible partitions, i.e many ways of subdividing a 3D dataset into blocks. The cost of finding out the optimal subdivision should not exceed the gain finally obtained by load balancing compared to a naive subdivision for which the cost is null. For finding out S we need to compute amounts of full bricks, which is very costly. Exploring all possible subdivisions to keep the best one, using for instance dynamic programming, is obviously too expensive. We propose to use a simple algorithm based on some heuristics along with a fast iterative procedure. The algorithm consists of two steps. In a first step, the dataset is split using a BSP-tree. In a second step, we iteratively move the splitting plane between blocks according to the values of $D(B_i,B_j)$ until we reach a local balanced state for the two corresponding blocks.

© The Eurographics Association 2012.

5.2.4. Computing the number of full bricks in a block

Computing the number of full bricks in a given block B_k is an operation ordered by the CPU to the different GPUs. As described in section 4, each GPU has already computed its own set of bricks (recall that bricking is performed in parallel on the GPUs). So each GPU is able to compute at least its own number of full bricks for this set (or for a part of it). Since a block B_k might contain bricks spread on different GPU(s), the computation must also be distributed accordingly. In other words, for each block for which we want to compute the number of full bricks, the CPU simply splits the computation on the corresponding GPU(s). To compute the partial number of full bricks on one GPU, we use an optimized reduction algorithm from the NVIDIA CUDA SDK [Har08] using shared memory within multi-processors. Determining the number of bricks is then done by copying the subtexture in which we want to count this number into a new memory area, and then perform a reduction on this memory area. Because our iterative algorithm moves a splitting plane in one or the other direction to add or remove bricks from blocks (see next two sections), the CPU can update the corresponding number of full bricks in an incremental way, which avoids redundant computations. In other words, the GPU re-computes only the number of full bricks for the added / removed parts. In fact, only the initial step (initial set of blocks) requires a full computation.

5.2.5. Splitting the dataset

The first step of our dataset subdivision technique consists in choosing an initial set of blocks $\{B_k | k \in [1,n]\}$ using a binary space partitioning tree (BSP). This is motivated by the fact that the BSP allows us to split data in the same direction multiple times, as opposed for instance to KD-trees. We also assume in the following that n is an integer power of two. At each node of the BSP tree, we split the corresponding dataset part into two equal subparts (in terms of number of voxels) using a median plane chosen among directions X, Y or Z. The root node corresponds to the entire dataset. We then recursively build the tree by splitting the successive blocks along a single direction. We choose the direction corresponding to the highest resolution in terms of voxels, i.e. for a block of resolution $(128 \times 64 \times 64)$ for instance, we will choose the X direction. This operation ends when we have a tree with n leaves. The choice of the highest resolution is motivated by the fact that it is likely to provide most flexibility in terms of splitting plane displacement (see second step). The subparts of the dataset corresponding to the leaves of the tree represent the initial set of blocks.

Given the heuristic we use, we make sure that the amount of full bricks is not too disparate for these blocks. This first step generally already allows us to coarsely load balance the computation for datasets that have an equiprobable distribution of full bricks.

5.2.6. Adjusting adjacent blocks

Based on the initial set of blocks and the corresponding BSP tree, the second step uses the criterion $D(B_i, B_j)$ to move the splitting plane between adjacent blocks B_i and B_j associated with two siblings in the BSP-tree. The goal is to converge towards a final set of blocks that better matches the desired equality of number of full bricks.

This process iterates over each internal node of the BSP-tree in a top-down fashion. It is repeated until the tree is considered balanced or a processing time limit is reached (fixed at 100ms in our case). At each node, we compute the load-balancing criterion $D(B_i,B_j)$, where B_i and B_j are the two blocks associated to its two child nodes. If $|D(B_i,B_j)|$ is larger than a threshold value, then the two blocks are considered unbalanced, and the splitting plane is moved according to the sign of D. This process is repeated until $|D(B_i,B_j)|$ is below the threshold or the sign of $D(B_i,B_j)$ changes.

5.2.7. Rendering

A static distribution allows to directly use a classical sort-last parallel direct volume rendering method. Partial rendering is performed on each device. We then arbitrarily mark a GPU as the *master GPU*. This GPU will then be in charge of recombining the partial images by sorting them according to the position of the data block (back to front) and merging them using alpha blending. This process is illustrated in Figure 3.

5.3. Dynamic distribution

With this second distribution scheme, we consider a classical producer-consumer approach. The dataset is split into fixed-size blocks. The number of blocks generated is strictly greater than the number of GPUs of the targeted computer and such that it entirely fits into the GPU memory. Note that the blocks must be enlarged by the size of the radius R_Ω. The blocks and their associated bricks are then dynamically distributed to idle GPUs for computing using a critical section. The Ambient Occlusion factors are then read back on the CPU side, which maintains the global array for computed factors. This strategy allows us to reduce the footprint of Ambient Occlusion computations as it uses smaller blocks for computation. It also allows us to get load-balanced results as the GPUs are always busy with computations: even if some blocks take more time for processing, others GPUs are still busy with the consumer-producer approach. This can be beneficial on heterogeneous architectures, where each GPU does not have the same computational power. However, if the size of blocks is chosen too small compared to radius R_Ω there will be an important data transfer overhead. Indeed, Ambient Occlusion needs all neighboring voxels inside the radius R_Ω for computation. If a block is composed of a single voxel for instance, the actual transfered data block must be a cube of width, height and depth $(2R_\Omega + 1)$.

© The Eurographics Association 2012.

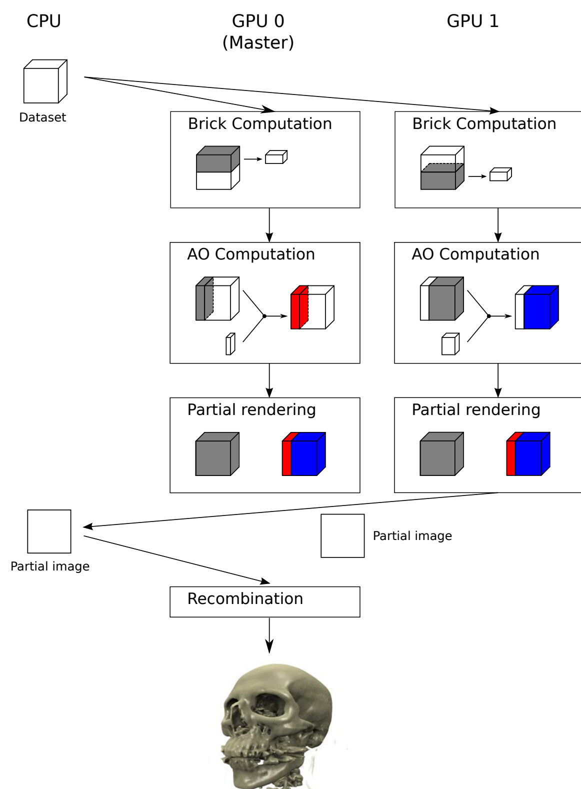

Figure 3: *Illustration of the multi-GPU rendering process.*

6. Results

All tests were performed on a computer with two hexacore Intel X5650, 72 Gb of Ram and 8 NVIDIA Tesla C2050 with 3071 Mb of inboard RAM. The parallel programming uses OpenMP on the CPU side and CUDA on the GPU side.

We conducted our tests on following datasets (see Figure 4): the Foot ($256 \times 256 \times 256$), the CT Knee ($379 \times 229 \times 305$), the Backpack ($512 \times 512 \times 373$), the Colon Phantom ($512 \times 512 \times 442$) and the Stag beetle ($832 \times 832 \times 494$). In the total times measured, we take into account the transfer times, the reduction processing times (for computing full bricks), the Ambient Occlusion computation times and the bricking times for each GPU. Regarding Ambient Occlusion parameters, we use an initial ray offset and step of 1 voxel and a sphere radius R_Ω of 32 voxels. For the bricking configuration, we use bricks with 4^3 voxels with an overlap of 1 voxel. Series of tests on our multi-GPU setup have shown that the 4^3 brick size results in most efficient computation times.

In the following, we present the impact of the block size on the computation time in the dynamic repartition. Then we analyze the performance and the scalability of our solution on the different GPUs.

Dynamic distribution – Figure 5 illustrates the timings measured for the dynamic distribution using bricking for different block sizes : from 32 to 256. We notice that the bigger the dataset gets, the more it becomes interesting to choose a greater block size, as demonstrated by the Stag beetle. Too small blocks imply a transfer overhead and less texture cache exploitation, as the number of blocks is more important and we have to append an R_Ω overhead to each of them. Too large blocks, on the other hand, result in load unbalance (i.e. more computation time), because the global number of blocks can reach the global number of GPUs. In such a case the static distribution scheme improves load balancing.

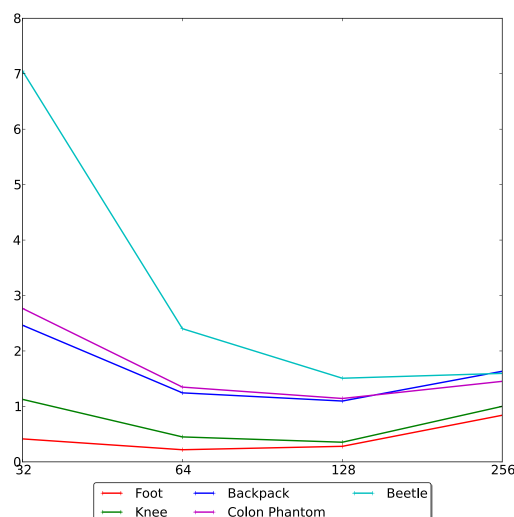

Figure 5: *Influence of the block size for the dynamic distribution given in sec.*

Parallel computation time analysis – Using the backpack dataset, Figure 6 shows the different computation times: single GPU without (a left) and with (a right) bricking, the naive subdivision on 8 GPUs without bricking (b), the naive subdivision using bricking (c) and finally the static distribution including load balancing (d). A first observation is that the time taken by the Ambient Occlusion computation (in orange) is the most costly part, respectively taking from (a) to (d) ~98%, 84% to 88%, 23% to 81% and 59% to 71% of the total time. The bricking step takes less than one percent (not even visible on the figure). The data transfer time participates in 1% to 12%, and the load balancing step takes 7% of the computation time in the last case. The second observation is that the use of bricking and load balancing do both improve performances. The global computation is decreased by 260 ms between a naive approach using no bricking (b) and the optimized one (d).

Table 1 shows computation times for different datasets. In each subtable, we present computation times using one GPU without bricking (reference time), naive static paralleliza-

© The Eurographics Association 2012.

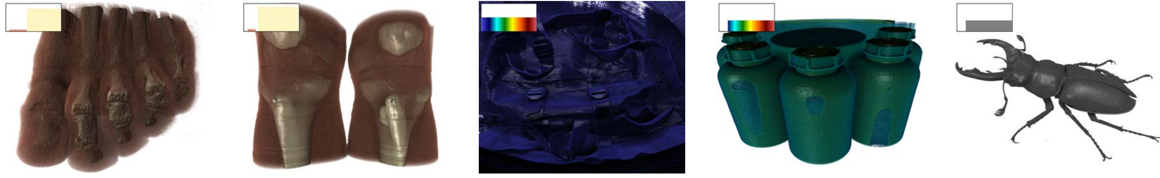

Figure 4: *Transfer functions for the different datasets used.*

tion, static parallelization with bricking, static parallelization with bricking and load-balancing and finally dynamic parallelization with bricking using blocks of size 128^3. The second column shows the different computation times in the following form $t_{total}\,(t_{min}, t_{max})$, where t_{total} is the computation time of the whole process, with the bricking and transfer times included, t_{min} and t_{max} are respectively the minimum and maximum computation times, but only for the raycasting process computing Ambient Occlusion, since this always represents the most costly part. The parallelized times are produced using 8 GPUs. We also presents speedups noted S in the last column. It is computed as the division of the total time taken using a single GPU over the current total time. Notice first that going from one GPU setup to either static or dynamic parallelization with 8 GPUs greatly reduces the computation times. However, the time reduction is not proportional to the number of GPUs in the computer as illustrated by the presented speedups. This can be explained in several points. First we add transfer times between the different GPU and the CPU. These transfer times are further increased by the fact that we need to take into account the Ambient Occlusion radius R_Ω when splitting the dataset into blocks. Bricking also substantially helps reducing computation times, although it also introduces some additional preprocessing. Nevertheless, the combination of bricking and multi-GPU allows to get super-linear speedups when comparing the mono-GPU implementation to either the parallelized and optimized static or dynamic versions for the presented datasets.

Scalability – Figure 7 illustrates the scalability of Ambient Occlusion precomputation times for the Colon Phantom dataset using an increasing number of GPUs. We also compare static and dynamic distribution with and without using bricking. The scalability is good for all techniques. Note that static and dynamic distributions lead to very similar performance. This is explained by the fact that the overhead resulting from the computation of load balancing in the static case (counting full bricks) seems to be similar to the overhead of data transfer related to the dynamic case (provided the blocks are sufficiently large).

	$t_{total}\,(t_{min}, t_{max})$	S
Foot (256 × 256 × 256)		
Single GPU	1.871 (1.847)	1.0
Static Naive	0.290 (0.221, 0.238)	6.45
Static Bricked	0.201 (0.055, 0.121)	9.30
Static Load-balanced	0.204 (0.090, 0.106)	9.17
Dynamic Bricked	0.280 (0.036, 0.197)	6.68
Knee (379 × 229 × 305)		
Single GPU	2.906 (2.809)	1.0
Static Naive	0.448 (0.341, 0.356)	6.48
Static Bricked	0.407 (0.115, 0.279)	7.14
Static Load-balanced	0.457 (0.187, 0.233)	6.35
Dynamic Bricked	0.354 (0.186, 0.225)	8.20
Backpack (512 × 512 × 373)		
Single GPU	10.235 (10.108)	1.0
Static Naive	1.458 (1.223, 1.285)	7.01
Static Bricked	1.341 (0.298, 1.095)	7.63
Static Load-balanced	1.197 (0.717, 0.857)	8.55
Dynamic Bricked	1.096 (0.662, 0.867)	9.33
Colon Phantom (512 × 512 × 442)		
Single GPU	12.794 (12.642)	1.0
Static Naive	1.815 (1.466, 1.648)	7.04
Static Bricked	1.654 (0.211, 1.373)	7.73
Static Load-balanced	1.289 (0.717, 0.920)	9.92
Dynamic Bricked	1.142 (0.804, 0.850)	11.20
Stag beetle (832 × 832 × 494)		
Single GPU	35.479 (35.038)	1.0
Static Naive	4.956 (4.322, 4.441)	7.15
Static Bricked	1.866 (0.317, 1.158)	19.01
Static Load-balanced	1.766 (0.317, 0.891)	20.09
Dynamic Bricked	1.508 (0.525, 0.608)	23.52

Table 1: *Comparison of computation times in seconds for different datasets on our 8-GPU setup. For each dataset, we show computation times for one GPU without bricking (reference) and our optimization strategies. Regarding to the times presented, the first number is the total time to compute Ambient Occlusion and the number in parenthesis are respectively the minimum and maximum computation times of the raycasting part only. We also present the associated speedups in the last column. They are computed using total times.*

7. Conclusion & Future works

We introduced and compared two multi-GPU data distribution strategies for efficiently computing Ambient Occlusion-

© The Eurographics Association 2012.

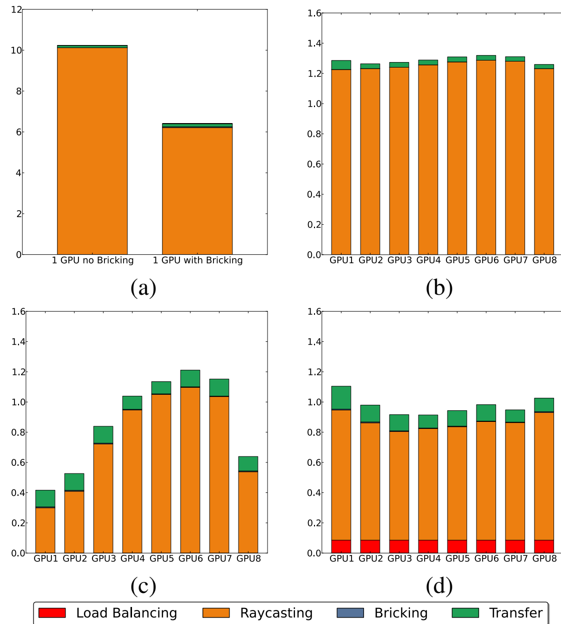

Figure 6: *Precomputation times in seconds for the Backpack dataset with Ambient Occlusion on 8 GPUs. Histogram (a) shows results on 1 GPU without and with bricking. Histograms (b) and (c) illustrate the static naive parallel Ambient Occlusion without and with bricking. The last one (d) shows Ambient Occlusion with bricking and load-balancing. The last line is the legend common to all histograms.*

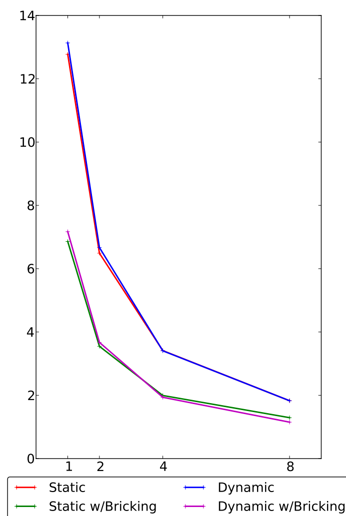

Figure 7: *Scalability of the static and dynamic distribution schemes for an increasing number of GPUs (in sec.).*

based shading in the context of parallel direct volume rendering. Ambient Occlusion is a time consuming shading technique that greatly improves data perception by adding soft shadows. Our setup allows us to reach interactive framerates on low end PCs equipped with multiple GPUs. The two distribution schemes take benefit of the use of a bricking technique. The static strategy proceeds in two steps. We first divide the dataset using a BSP-tree and then refine the load-balancing by moving the previously computed splitting planes. The dynamic strategy consists in breaking the computation into small blocks and distribute them to available GPUs using a producer-consumer approach. Using five datasets, we have tested both distribution strategies for up to 8 GPUs. The results show a good scalability as the speed up is almost linear. Both strategies are also fully compatible with a classical parallel sort-last rendering algorithm.

Regarding future works, we would like to investigate asynchronous transfers to overlap computations. An important point will consist in experimenting this method on a multi-GPU cluster and on heterogeneous setups (different GPUs with different computational power). Moreover, the evolution of GPU technology could give more opportunities to further optimize Ambient Occlusion algorithms.

References

[ADM10] Ancel A., Dischler J.-M., Mongenet C. : Feature-driven ambient occlusion for direct volume rendering. In *International Symposium on Volume Graphics* (may 2010), IEEE/EG. 2, 4

[CM09] Correa C. D., Ma K.-L. : The occlusion spectrum for volume classification and visualization. *IEEE Transactions for Visualization and Computer Graphics 15*, 6 (October 2009). 2

[DE07] Desgranges P., Engel K. : Fast ambient occlusion for direct volume rendering. In *United States Patent Application* (2007), no. #20070013696. 2

[FCS*10] Fogal T., Childs H., Shankar S., Krüger J., Bergeron R. D., Hatcher P. : Large data visualization on distributed memory multi-gpu clusters. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 57–66. 2

[Har08] Harris M. : Optimizing parallel reduction in cuda. White paper, NVIDIA, 2008. 6

[HLY07] Hernell F., Ljung P., Ynnerman A. : Efficient ambient and emissive tissue illumination using local occlusion in multiresolution volume rendering. In *Eurographics/IEEE-VGTC Symposium on Volume Graphics* (2007), pp. 1–8. 2, 3

[HLY10] Hernell F., Ljung P., Ynnerman A. : Local ambient occlusion in direct volume rendering. *Visualization and Computer Graphics, IEEE Transactions on 16*, 4 (july-aug. 2010), 548 –559. 2

[LB00] Langer M. S., Bülthoff H. H. : Depth discrimination from shading under diffuse lighting. *Perception 29* (2000), 649–660. 2

[MMD08] Marchesin S., Mongenet C., Dischler J.-M. : Multi-gpu sort last volume visualization. In *EG Symposium on Parallel Graphics and Visualization (EGPGV'08)* (april 2008), Eurographics. 2

© The Eurographics Association 2012.

[MR10] MESS C., ROPINSKI T. : Efficient acquisition and clustering of local histograms for representing voxel neighborhoods. In *IEEE/EG International Symposium on Volume Graphics* (2010), Westermann R., Kindlmann G., (Eds.), Eurographics Association, pp. 117–124. 2

[MSM10] MARTIN S., SHEN H.-W., MCCORMICK P. : Load-balanced isosurfacing on multi-gpu clusters. In *EGPGV '10 : Proceedings of Eurographics Symposium on Parallel Graphics and Visualization 2010* (May 2010), pp. 91–100. 2

[Pho75] PHONG B. T. : Illumination for computer generated pictures. *Commun. ACM 18*, 6 (1975), 311–317. 1

[RBV*08] RUIZ M., BOADA I., VIOLA I., BRUCKNER S., FEIXAS M., SBERT M. : Obscurance-based volume rendering framework. In *Volume and Point-Based Graphics 2008* (aug 2008), pp. 113–120. 2

[Rit07] RITSCHEL T. : Fast GPU-based Visibility Computation for Natural Illumination of Volume Data Sets. In *Short Paper Proceedings of Eurographics* (Sept. 2007), pp. 17–20. 2

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

[RSKU*10] RUIZ M., SZIRMAY-KALOS L., UMENHOFFER T., BOADA I., FEIXAS M., SBERT M. : Volumetric ambient occlusion for volumetric models. *Vis. Comput. 26*, 6-8 (June 2010), 687–695. 2

[Sal07] SALAMA C. R. : Gpu-based monte-carlo volume raycasting. In *PG '07 : Proceedings of the 15th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 411–414. 2

[ŠPBV10] ŠOLTÉSZOVÁ V., PATEL D., BRUCKNER S., VIOLA I. : A multidirectional occlusion shading model for direct volume rendering. *Computer Graphics Forum 29*, 3 (june 2010), 883–891. 1, 2

[SPH*09] SCHOTT M., PEGORARO V., HANSEN C. D., BOULANGER K., BOUATOUCH K. : A directional occlusion shading model for interactive direct volume rendering. In *Computer Graphics Forum* (2009), vol. 28. 1, 2

[Ste03] STEWART A. J. : Vicinity shading for enhanced perception of volumetric data. In *Proceedings of the 14th IEEE Visualization Conference* (2003), pp. 355–362. 2

[ZIK98] ZHUKOV S., IONES A., KRONIN G. : An ambient light illumination model. In *Rendering Techniques* (1998), pp. 45–56. 2

© The Eurographics Association 2012.